

AD-A223 719

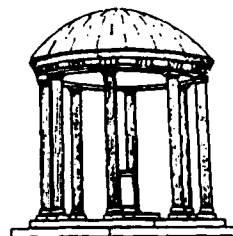
The COOL Library: Supplementary Documents, Version 1.0

CO01

DTIC
ELECTE
JUN 11 1990
S D D

James Coggins

SoftLab Software Systems Laboratory
The University of North Carolina
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

**DISTRIBUTION STATEMENT A**

Approved for public release
Distribution Unlimited

*Supported in part by the NASA Center of Excellence in Space Data and Information
Sciences, contract#500-62 and the Office of Naval Research,
contract#N-00014-86-K-0680.*

Reference and title

CO01

The COOL Library: Supplementary Documents, Version 1.0

Publication history

Version 1.0: October 1989

Copyright and trademarks

Copyright ©1989. University of North Carolina at Chapel Hill. All rights reserved. Copying without fee is permitted provided that the copies are not made or distributed for direct commercial advantage, and credit to the source is given. To copy otherwise, or republish, requires a fee and/or specific permission.

Authors

James Coggins

The COOL Library

Supplementary Documents

Table of Contents

| | |
|---|------|
| 1. Introducing COOL..... | 1-1 |
| 1.1 What is COOL? | 1-1 |
| 1.2 Motivation for COOL..... | 1-2 |
| 1.2.1 Software requirements for image science research..... | 1-2 |
| 1.2.2 Incipient Software Crisis..... | 1-4 |
| 1.2.3 Software Engineering Support | 1-4 |
| 1.2.4 Separation of Concerns | 1-5 |
| 1.3 Structure of the COOL Documentation..... | 1-5 |
| References..... | 1-6 |
| 2. Object-Oriented Design | 2-1 |
| 2.1 Object-Oriented Design Conventions..... | 2-1 |
| 2.2 The Evolution of Object-Oriented Design | 2-2 |
| 2.3 Encapsulation | 2-3 |
| 2.4 Inheritance..... | 2-6 |
| References..... | 2-6 |
| 3. Object-Oriented Programming in C++..... | 3-1 |
| 3.1 From C to C++..... | 3-1 |
| 3.2 What You Already Know Probably Works | 3-1 |
| 3.2.1 C++ has strong type checking..... | 3-2 |
| 3.2.2 Double-slash is a comment delimiter | 3-2 |
| 3.2.3 Declarations may appear anywhere..... | 3-3 |
| 3.2.4 Variables and arguments can be declared const..... | 3-3 |
| 3.2.5 Parameter lists use ANSI C syntax | 3-3 |
| 3.2.6 Streams provide simpler or customized I/O | 3-4 |
| 3.2.7 Arguments can be passed by reference..... | 3-4 |
| 3.2.8 Storage management is simplified | 3-5 |
| 3.2.9 Inline functions speed execution..... | 3-7 |
| 3.2.10 Overloaded functions and operators are supported | 3-7 |
| 3.3 C++ Support for Encapsulation..... | 3-9 |
| 3.3.1 Classes Define the Structure and Function of Objects..... | 3-9 |
| 3.3.2 Constructors and Destructors Create and Destroy Objects .. | 3-10 |
| 3.3.3 Private Data and Procedures Define Internal Structure | 3-12 |
| 3.3.4 Public Member Functions Define Externally Visible Structure | 3-13 |
| 3.3.5 Programs use classes as if they were predefined types | 3-14 |
| 3.4 Using Inheritance | 3-15 |
| 3.4.1 Declaring a subclass..... | 3-15 |
| 3.4.2 Accessing Members of the Superclass..... | 3-16 |
| 3.4.3 The Semantics of Inheritance | 3-16 |
| 3.4.4 Member Functions Most Classes Need..... | 3-19 |
| 3.4.5 Virtual Functions..... | 3-20 |
| References..... | 3-21 |

STATEMENT "A" per Dr. A. Van Tilborg
 ONR/Code 1133
 TELECON

6/11/90

VG

| | |
|------|----------------------|
| Jes | |
| Dist | Avail and/or Special |
| A-1 | |



| |
|-------------------------------------|
| <input checked="" type="checkbox"/> |
| <input type="checkbox"/> |
| <input type="checkbox"/> |

per call

| | |
|---|------|
| 4. Designing C++ Libraries..... | 4-1 |
| 4.1 The Challenge of Software Library Design | 4-1 |
| 4.2 Why comprehensive, monolithic hierarchies are a bad idea | 4-2 |
| 4.3 Why a toolkit of tiny classes is a bad idea..... | 4-3 |
| 4.4 Why UI-controlled implementation is probably not a great idea | 4-4 |
| 4.5 Objectives of Library Design | 4-5 |
| 4.6 Library Design in Practice | 4-6 |
| 4.7 Conclusion: Toward a Collaborative Class Design Methodology..... | 4-7 |
| References..... | 4-8 |
| 5. Managing C++ Libraries..... | 5-1 |
| 5.1 The Problems of Library Management..... | 5-1 |
| 5.2 Directory Management..... | 5-2 |
| 5.2.1 Library Directory Organization | 5-2 |
| 5.2.2 Separating Member Functions..... | 5-3 |
| 5.3 Dependency Management..... | 5-4 |
| 5.3.1 Dependencies..... | 5-5 |
| 5.3.2 Rejected Approaches..... | 5-6 |
| 5.3.3 Proposed Solution | 5-7 |
| 5.3.4 Dependency Files..... | 5-7 |
| 5.3.5 The Prelude File..... | 5-8 |
| 5.3.6 Using the System..... | 5-9 |
| 5.3.7 Robustness | 5-9 |
| 5.4 Optimizations..... | 5-10 |
| 5.4.1 Size Optimization | 5-10 |
| 5.4.2 Minimizing Recompilations | 5-11 |
| 5.5 Conclusion..... | 5-12 |
| References..... | 5-12 |
| Appendix A | 5-14 |
| 6. Implementation Techniques in COOL | 6-1 |
| 6.1 Managing Massive Data Structures..... | 6-1 |
| 6.2 Implementing Type Independence | 6-2 |
| 6.3 Encapsulating Device Dependencies | 6-3 |
| 6.4 Process Encapsulations..... | 6-4 |
| 6.5 Class Interface Encapsulations..... | 6-5 |
| 6.6 Separation of Concerns..... | 6-6 |
| 6.7 Conclusion: Is C++ Really the Right Tool?..... | 6-6 |
| 7. Program Examples | 7-1 |
| 7.1 A Simple Program Using COOL..... | 7-1 |
| 7.2 Preparing a Program Using COOL..... | 7-3 |
| 7.3 Displaytool | 7-4 |

Acknowledgements

The COOL library and the documentation for it has been one of my principal research activities for the three years since I arrived at the University of North Carolina. When I moved to Chapel Hill, I lost overnight the entire research software library I had been developing and using due to incompatibilities between the language and computing environments I had worked in for eight years and the UNIX environment at UNC. Many graduate students and faculty colleagues helped me to learn C++ and to begin building the library and its documentation. I am pleased to acknowledge their contributions here:

Tim Rentsch introduced me to the "lunatic fringe" of Object-Oriented Design and helped with some of the initial, critical decisions underlying some of the best innovations in the library. Jonathan Walker pointed the way toward complex but innovative and effective directory and makefile organizations and extolled some of the virtues of C++ to me when I was but a novice. Brice Tebbs explored and taught me basics of C++. Murugappan Palaniappan implemented and reimplemented many of the early classes as my design concepts refined through experience. His work appears in many places in the library, often in places other than where he wrote them. John Rohlf wrote the first version of the graphics classes. Andrew Glassner helped me to debug some of the graphics procedures by teaching part of our intro graphics course using a rapidly evolving COOL. The patience of the students in that class also deserves mention as COOL went through a rapid evolution (revolution?) between their assignments using the library. Greg Bollella's work through our department's Softlab to help me set up the administrative structures of COOL resulted in publication of our approach both on the Internet and in SIGPLAN Notices; this paper is substantially reproduced in Chapter 4 of this document. Greg's creative expertise with Makefiles will be further illustrated in a later public release of COOL in which the Makefile structure will support multiple architectures. Cathy Vishnevsky and James Chung did their required Writing Projects on C++ and COOL, respectively, and some portions of the chapter on C++ have been influenced by their reports.

My local users group, a captive audience whose sweat and time have been spent exploring, using, and sometimes working around COOL have earned a mention here, too: K. C. Low, Eric Fredericksen, Lisa Baxter, Yuchin Fu, Amandeep Jawa have been and are yet exploring the use of the library in our research. Help, comments and suggestions of various kinds have been received from Jonathan Leech, Graham Gash, Tim Cullip, John Gauch, Brian White, Russell Taylor, Michael Kelley, Steve Pizer, and Fred Brooks.

When I interviewed for a faculty position at UNC, Fred Brooks asked me how my computer vision research might shed any light on the problems of software engineering. I didn't have a very good answer then. Fred, I would like hereby to amend that response.

1. Introducing COOL

1.1 What is COOL?

COOL is a software library written in the object-oriented programming language C++ [1,2]. COOL was developed using the AT&T Cfront 1.2.1 compiler, but it should be compatible with other versions of C++. COOL has been developed with a "pure" object-oriented programming idiom; the encapsulation and inheritance features of C++ are exercised to the fullest extent possible. (An example of this policy is that "friend" functions and classes that disable the protections and restrictions of the object-oriented programming discipline are avoided.) COOL does not provide a monolithic object hierarchy as provided by the languages Smalltalk [3] and Objective-C ICpack 201 [4] or by the NIH (OOPS) library for C++ [5]. Instead, COOL consists of a group of separate inheritance hierarchies whose structure is optimized for the inherent dependencies of the concepts they represent in the target applications.

COOL contains class definitions relevant to research in computer vision, image pattern recognition, and computer graphics, which we will henceforth refer to as the "image sciences". Image science algorithms implement mappings between images and models. (Figure 1). Image processing algorithms map an input image to another image that is an enhanced, restored, compressed, or corrected version of the input image. Computer graphics algorithms take a model of a virtual world and a viewing specification and produce an image as output. Image pattern recognition algorithms infer from images some properties of a virtual world - often a simple measurement or classification is all that is required, not a complete virtual world model.

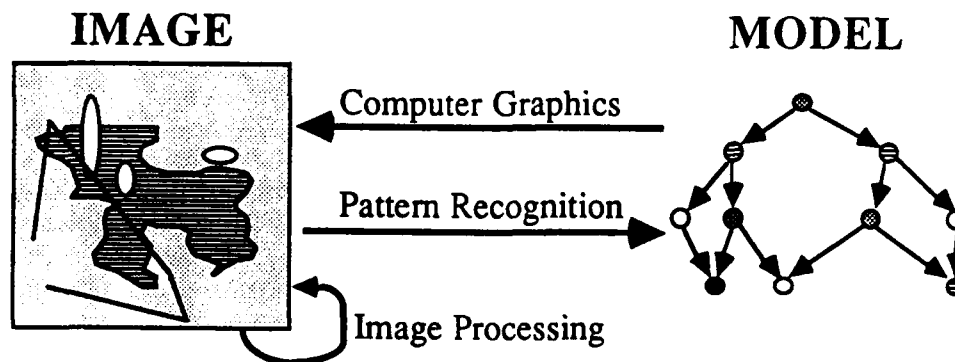


Figure 1: Relationships among the image sciences

COOL is likely to be of some interest for its unique functionality supporting image science research but even more for its coding and organization techniques. COOL's `nsqplib` sublibrary encapsulates matrices, points, vectors, and patterns. The `graphlib` sublibrary contains classes for generating and manipulating graphical world models and display lists. The `disklib` sublibrary

encapsulates disk file formats and operations. Classes are provided for random variates, subscripts, histograms, timers, and analog devices. These and other COOL structures may prove useful in themselves. The library organization and coding techniques illustrated in COOL will probably be of broader interest than the specific classes and implementations in COOL; therefore, the rationale for many of these features and the tradeoffs they embody are described in detail.

1.2 Motivation for COOL

COOL arose from the convergence of four notions: First, research in the image sciences requires a large software infrastructure that requires ongoing maintenance due both to changes in hardware support and to algorithmic advances. Second, an incipient software crisis pervades the image sciences because the body of knowledge and the supporting infrastructures are changing more rapidly than the software systems can keep up, and the research is advancing beyond the capability of existing software systems to support it. Third, recent advances in software engineering methodology, particularly object-oriented programming and languages for supporting it, can help to solve, or at least manage, the software problems facing the image sciences. Fourth, the software supporting the image sciences should adhere to a new design criterion: it needs to separate the concerns of (1) processing, (2) storage and communication, and (3) display and interaction. The rest of this chapter elaborates on these ideas.

1.2.1 Software requirements for image science research

Software for supporting the image sciences must address four issues: integration, data complexity, algorithm complexity, and device dependencies.

Integration

A collection of all known image science procedures does not constitute a solution to *any* image science problem. Integration of the methods is required both at the software level, so that the methods can work well together, and at the theory level so that different methods that are essentially the same can be considered together and compared on their theoretical power as well as on pragmatic issues such as run time. Integrating image science concepts at both of these levels requires a unified theoretical foundation that has been lacking in the image sciences. The unification and weeding of the diverse methods used in the image sciences may finally occur in the forging of integrated software systems. COOL is an experiment toward that end.

Data complexity

Image data is massive. While this data begins its existence as a spatial array of intensities, interest in spectrally and temporally sampled imagery is increasing, requiring manipulation of 3-D, 4-D, and higher-dimensional representations. In addition, the representation of the inferences obtained from images requires complex data structures such as multidimensional trees and graphs. Software for image pattern recognition must facilitate the transformation of data from one

form to another and support modifications of the forms in response to automatic processes or interactive manipulations. Graphical models often begin as complex data structures representing a virtual world and eventually yield a high-resolution image array. Complex data structures are used to enhance computational speed, flexibility, and interactivity. In view of this growing data complexity, it is critical that software for image science research employ all of the tools available for hiding irrelevant details at each implementation layer. The encapsulation facilities of object-oriented languages are well-suited to supply this need.

Algorithmic Complexity

Image science algorithms are complex, often requiring manipulation of multiple representations at once and involving implementations of sophisticated mathematical and computational methods. The details of the implementations are significant because any operation, when applied to such massive data, imposes a significant run-time cost; therefore, representations and forms of algorithms that gain speed at the cost of complexity cannot be ruled out of the question. Similarly, the order in which operations are performed may have a dramatic effect on the memory requirements of the process being executed. The flexibility to judge these tradeoffs must be left to the user (image science researcher) and must not be made *a priori* by the library developer. In addition, image science research often requires long run times, so most methods must be executable with no interactive user interface; backgrounding batch mode jobs will *always* be necessary because as our computer power increases we will want to apply our algorithms to ever larger data sets or to more sophisticated manipulations rather than using the improved hardware performance just to give faster compute times. Thus, the level of abstraction of the routines provided by the library must be chosen carefully. Process encapsulations must be provided both for well-defined, self-contained operations and for operations that are subjects of ongoing research and development.

Device dependencies

The devices for supporting the image sciences — cameras, digitizers, and displays — not to mention fast processors — are changing so rapidly that much of the effort devoted to maintaining image research libraries is devoted to updating the software to accommodate new devices. This requires that device independence be an important criterion in designing a software system for supporting the image sciences. Unfortunately, the trend in software toward window-oriented bit-mapped displays favors higher interdependence of the software and the user interface. This trend must be resisted in image science libraries in order to avoid massive rewrites as each new device, windowing system, or user interface library comes available. By abstracting and isolating the minimal, essential components of the user interface, COOL makes conversion between windowing systems easy.

For example, COOL was begun when SunWindows [6] was the dominant windowing system in our laboratory. Only two user interface classes were constructed, *imagetool* for SunWindows, and *ikonas* for the Adage/Ikonas 3600 display system. These classes support only the fundamental, abstract functions

required for the image sciences: primitive image display and line drawing, but now that the X Window System [7] is becoming a laboratory standard, very little of COOL needs to be rewritten. The `Ximagerool` class was added to COOL in a day.

Furthermore, by encapsulating communication protocols used to communicate between processors, COOL encapsulates tiresome details and provides minimal, easy-to-use interfaces for programmers to take advantages of networks of processors.

1.2.2 Incipient Software Crisis

A second aspect of the motivation for COOL is the incipient software crisis facing the image sciences. The range of applications now appearing is combining with the lack of a unified theoretical foundation to produce a fragmentation of the field that has inhibited progress while consuming great quantities of effort from research faculty and students. The combination of changing hardware and broadening applications has already stretched to the limit the software foundations in use. The older imaging library in our lab lacks data structures for the hierarchies of image regions we now find essential to our research. The library is not equipped to handle images of 1Kx1K pixels at all, and manipulation of five-dimensional image data (three of space, one of time, and one of scale) would require redesign of much of the library. Furthermore, the size of existing libraries has increased to the limit of the ability of their programmers to understand them. My own FORTRAN library, which I was working with before beginning to build COOL, was so large and involved so many encoded subroutine names that I was losing track of the tools I had formerly written to make later coding easier. Object-oriented design can help image scientists to organize their code around the important ideas of the field and hide irrelevant detail at each layer of abstraction, resulting in more effective software support for the research.

1.2.3 Software Engineering Support

Object-oriented design [8,9,10] is a set of conventions that address several aspects of the software crisis in the image sciences. Explicit support for encapsulation helps programmers to organize and package their code. Inheritance makes possible reuse of existing code and eliminates the need for error-prone and difficult-to-debug features such as arrays of pointers to functions. Operator overloading removes the exhausting concern with subroutine name encodings since the language processor distinguishes procedures by their names and argument lists jointly. Elimination of some syntactic restrictions makes writing readable code easier. All of these features are addressed in C++ and are exploited in the design and realization of COOL. In addition, we have developed a system of subdirectories and Makefiles that simplify the management and use of the library while bringing an object-oriented methodology to the administrative concerns of library management as well. Object-oriented design will make image science software easier to generalize and modify while organizing the software to hide irrelevant details and reuse critical, specialized code.

1.2.4 Separation of Concerns

One of the most powerful ideas in software engineering is the concept of *separation of concerns*. This notion says that in order to simplify the development of large programming systems, it is desirable to define modules so as to minimize the communication required among modules and thereby minimize communication overhead among their developers [11,12]. This criterion is optimized by defining separate domains of concern for each module and by separating the concerns of the architecture, implementation, and realization of each module. These separations of concerns can be enhanced by formal methods of design, by object-oriented programming structures such as C++ classes, and by appropriate design criteria for decomposing a large system into modules. COOL creates and exploits just such separations of concerns. COOL uses C++ classes to separate concerns of architecture from concerns of implementation and realization, customized makefile and header file structures to separate the concerns of coding from concepts of library management, and a set of C++ inheritance hierarchies to separate the concerns of processing, display and interaction, and storage and communication. These ideas will be elaborated upon in subsequent chapters.

1.3 Structure of the COOL Documentation

In order to provide a fairly complete description of COOL as well as the underlying coding idiom and organizational techniques, each document in this report will address a different innovation in COOL. Descriptions of the specific classes in COOL are provided in a separate report, The COOL Library: User's Manual.

Chapter 2 is an introduction to the discipline of Object-Oriented Design, including an overview of its historical development and a description of its principal features.

Chapter 3 introduces the C++ programming language and discusses how object-oriented programming is carried out in C++.

Chapter 4 discusses design criteria for C++ libraries, criticizing some commonly used (or assumed) criteria and illustrating a new set of criteria developed during the implementation of COOL.

Chapter 5 explains the nuts-and-bolts of establishing, organizing, and maintaining a C++ library. These organizational techniques are important because of the extra overhead in header files required to use C++ classes. The scheme described supports good software engineering practices, requires minimal effort to maintain, invades minimally on the programmer using the library, and is flexible enough to handle easily special cases such as mutual dependencies and maintenance of the library for multiple architectures.

Chapter 6 describes the programming idiom used in COOL, including several different types of encapsulation used in the library.

Chapter 7 describes how to use COOL and includes some example programs.

References

1. Stroustrup, Bjarne, *"The C++ Programming Language"*, Addison-Wesley, 1986.
2. Lippman, S. B., *C++ Primer*, Addison-Wesley, 1989.
3. Goldberg, A and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley 1983.
4. Stepstone Corporation, *ICpack 201 Tutorial Examples Manual*, Stepstone, 1988.
5. Gorlen, K. "An Object-Oriented Class Library for C++ Programmers", *Software: Practice and Experience*, vol. 17, no. 12, December 1987, p. 899.
6. Sun Microsystems, *SunView Programmer's Guide*, Sun Microsystems, 1986.
7. Nye, A., *Xlib Programming Manual for Version 11 of the X Window System*, O'Reilly and Associates, 1988.
8. Cox, B. J., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.
9. Rentsch, Tim, *"Object-Oriented Programming"*, SIGPLAN Notices, vol. 17, no. 9, September 1982, pp. 51-57.
10. Stroustrup, Bjarne, *"What is Object-Oriented Programming?"*, Proc. 1st European Conference on Object-Oriented Programming, Paris, 1987. In
11. Parnas, D. L, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, Dec 1972, pp. 1053-1058.
12. Parnas, D. L, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, March 1976, pp.1-9.

2. Object-Oriented Design

Object-Oriented Design refers to a set of conventions for organizing large software systems. Language support for object-oriented design includes facilities for data abstraction and inheritance.

2.1 Object-Oriented Design Conventions

Object-Oriented Programming has been described as being "in the 1980's what 'structured programming' was in the 1970's" [1]. Advertising hype and overclaiming by enthusiasts of the approach have made "object-oriented" little more than a high-tech synonym for "good". Thus, one sees all kinds of things labelled as being "object-oriented" or supporting "object-oriented methods" whether or not they really have anything to say about the programming methodology. The parallel with the way the term "structured programming" was used in the 1970's and early 1980's is striking.

Structured Programming is a coding convention whose application yields low-level code structures that are easier for people to understand (and therefore easier to debug and maintain). Reasoned arguments for adopting Structured Programming involve measures of code complexity, the applicability of formal and informal verification methodologies, the theoretical power of various control constructs, and anecdotal evidence concerning the understandability of code. Structured Programming became a serious undertaking when it was proved that any program can be written using only a small set of control structures: the set {sequence, iteration, and conditional} yields programming languages such as Pascal, C, and Algol; the control set {composition of functions, recursion, and conditional} yield Lisp and its derivatives. Additional (quite entertaining) philosophical debates have concerned the appropriate use of the `goto` construct and the value of various alternative control structures.

Object-Oriented Programming is a code packaging convention that allows a designer to impose a reasonable structure on large software systems based on the notions of encapsulation and inheritance. This convention provides several useful software development innovations. First, it provides a useful operational definition of module (a class is a module). Second, it establishes an organizing principle for task decomposition (minimizing communication and dependencies among modules). Third, it enforces a useful formal separation between architecture, implementation, and realization (implemented in the separation between public and private members of a class and in the separation of class declaration from class realization). Object-oriented design simplifies the structure of large software systems just as a decade ago structured programming simplified the structure of code segments. In addition, object-oriented design separates and clarifies some system design responsibilities, making management of software teams more above-board and straightforward, and gives us some new tools for thinking about software design.

It is possible, in principle, to adopt object-oriented design for systems written in languages that do not directly support object-oriented constructs. As with structured programming, however, there are great advantages in having a language embody and enforce the conventions. C++ supports object-oriented programming without abandoning the familiar notation and environment of C [2].

Object-Oriented Design is a good and useful methodology, but it is not the "silver bullet" that will kill our software demons [3]. Object-oriented design does not address the essential problems of software design: figuring out what you really need to do, figuring out how to do it, figuring out what you failed to see the first time and iterating. Object-Oriented design does address several substantial incidental issues that arise in project management and design. The advantages of Object-Oriented Design are significant, but they do not change the nature of the software development process. On this point, I disagree with those who claim that object-oriented design is a fundamental conceptual leap [1]. Instead, I see this new methodology as the latest step in the natural evolution of the technologies and methodologies for programming.

2.2 The Evolution of Object-Oriented Design

Object-Oriented Design is the latest product of a long history in which technological developments in programming languages have led to improved programming methodologies [4]. Each new methodology was, in its day, heralded as a great conceptual breakthrough, especially by those who stood to profit by the acceptance of the new methodology. Thus, the hype accompanying object-oriented design also has a long tradition. In the following paragraphs we review high points of this history to show that Object-Oriented Design, far from being a conceptual breakthrough, is actually a normal evolutionary development of programming methodology.

Assemblers made symbolic naming possible, especially symbolic addressing, which led directly to relocatable code. The programming methodology called procedural programming was made possible by the development of closed subroutines. The ability to isolate name spaces and to explicitly indicate information transfers through argument lists permits a partial separation of concerns between implementors and users. The convention of data hiding, supported by the technical innovation of object libraries, allows a partial separation of concerns between architects and implementors but provides little flexibility for users.

Data abstraction extends the data hiding concept by providing support in the programming language for abstract or user-defined data types. With data abstraction, the organizing principle for developing large systems is encapsulation: one defines a complex collection of data as a single abstract data type under a single name, providing a higher level of abstraction. Now the entire collection of data can be referred to (as in procedure arguments) by a single name.

One defines the types so as to minimize the interdependencies among them. The abstract data type becomes the operational definition of a module, supporting a nearly complete formal separation of concerns between users, architects, and implementors.

Structured Programming and the associated stepwise refinement design method were brought to general attention at about the same time as was data abstraction. All of these ideas were introduced and implemented in the design (and marketing) of the programming language Pascal.

The term "object-oriented programming" is often used as a synonym for data abstraction, especially when the procedures that operate on the data are allowed as members of the abstract data type, but the term properly refers to the use of class hierarchies and inheritance to make explicit the commonalities among abstract data types. (Note: Ada is not an object-oriented language!) Classes also allow data and the procedures that operate on the data to be encapsulated together, providing a yet higher degree of abstraction for system management and integration, but this is an enhanced aspect of data abstraction, not an aspect of object-oriented design. The implementation of the language Smalltalk took the concepts of object-oriented design to their logical extreme. Smalltalk operates in a rich, interpreted environment that includes powerful tools for examining and modifying system code. This interpretive environment dramatically impacts the way one pursues one's goals and makes True Believers out of Smalltalk users. The effect of having such a rich programming environment is even more immediate and dramatic than Object-Oriented Design itself. The interpretive environment of Smalltalk is a remarkable innovation, but it is not inseparable from Object-Oriented Design.

2.3 Encapsulation

When using object-oriented design, logically related data and operations are packaged together (encapsulated) in a *class*. An instance of a class is called an *object*. The class definition specifies both the memory structure and the set of allowed operations for objects of the class. The operations and data in a class may be *private*, so only objects belonging to the same class may access them, or *public*, so objects of any class can use them. (C++ also supports an intermediate level of isolation called *protected*. Protected components may be accessed by objects of derived classes only.)

The support for public and private components creates an explicit separation of the concerns of users, architects, and implementors. The public definition of a class is a contract between the users of the class and its architects that certain functionality will be supported. The collection of class declarations in a library constitutes a contract between the architect and the implementor that certain relationships among the classes will be supported. The implementation of the class structures is then the implementor's domain. This separation of concerns focuses the negotiations between users and architects on how the class structure

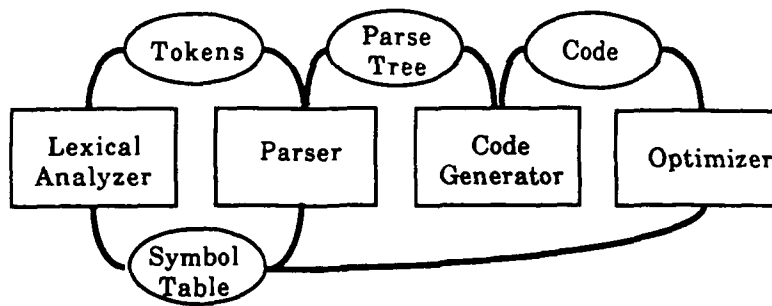
can be defined to faithfully reflect the user's mental model of the objects being represented, which is exactly where their discussion should be centered.

The operations (*methods, member functions*) defined for a class are invoked on an object by sending a *message* to the object (or invoking a member function of the object). The message names what is to be done by the object but does not specify how it is to be done; "how" questions are reserved to the implementor. In particular, the implementor may write the code for the required function in the member function of the class, or the processing of the message may be deferred to other objects by simply sending messages to request appropriate actions by those objects. Class libraries often resemble bureaucracies: sometimes the purpose of a class is simply to route messages to objects of other classes.

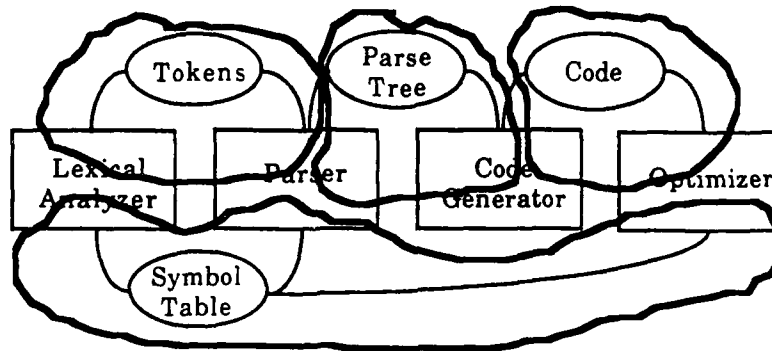
Constructors and destructors are special messages that are invoked automatically when an object is created (by declaration or by explicit allocation using the `new` operator) or destroyed (by exiting the scope of the object or by explicit deallocation using the `delete` operator) to ensure that the object is initialized or deallocated correctly. The explicit inclusion of these essential operations gives the class definition a consistent and complete structure. In C++, constructors and destructors are explicitly defined in a uniform fashion (the constructor has the same name as the class; the destructor name is the class name preceded by a tilde).

The analogy between classes and objects on the one hand and types and variables on the other hand is obvious: classes and types are declarations that are instantiated by objects and variables, respectively. However, the object-oriented programming constructs are more than a renaming of familiar ideas. The significance of classes can be compared to the significance of the Pascal `record` or the C `struct`. Records and structs allow logically related data items of different types to be encapsulated together under a single name and then hierarchically organized. This encapsulation serves both to agglomerate related items and to separate unrelated items. The language support for these constructs enforces the conventions defined in the record or struct declarations. Classes provide a similar organization for data and code jointly. Thus, the class structure organizes the software system and the data structures simultaneously.

To illustrate how data abstraction and encapsulation affect problem decomposition, consider the modularization of a simple compiler (Figure 2). The compiler will operate in four passes: lexical scan, parsing, code generation, and optimization. A decomposition of the compiler project that assigns one team to each pass (Figure 2a) imposes a heavy communication load between the teams. For example, the teams must jointly design the symbol table, token list, parse tree, and code list data structures. In an object-oriented design (Figure 2b), class definitions for these objects would be the principal modules, and each module would be written by one team. Negotiations between the teams can then focus on the functionality and user interfaces of the modules; the internal structures and algorithms are hidden inside the modules and those internal structures are explicitly under the control of a team that specializes in their implementations.



(a)



(b)

Figure 2: Two module decompositions of a compiler

(a): Traditional process-based decomposition

(b): Object-Oriented decomposition

Once the basic classes are designed, additional classes can be created for the lexical scanner, parser, code generator, and optimizer. These *process encapsulations* focus on controlling interactions among the objects involved in each pass without the distracting details of internal representations, operation, and naming of the basic objects. Objects that have as their purpose the control of interactions among other objects are called *enzymes* or *catalytic objects*.

The compiler example illustrates how data abstraction clarifies several practical issues in the management of large software projects. The distinction between public and private data and operations provides a useful separation of concerns between architecture, implementation, and realization. The class structure focuses discussion among users and architects on the public structure of the classes and how this public structure can best reflect the mental constructs they represent. The ability of object-oriented design to focus debate at a high level is one of the most significant contributions of the methodology.

Encapsulation is familiar to most programmers who have worked with C or Pascal. The additional aspects of encapsulation provided by object-oriented languages, the ability to encapsulate operations with the data they operate on or to provide groups of operations under a single name, are usually easy to assimilate

and use in one's own system design almost immediately. Inheritance, on the other hand, is much more subtle and difficult to learn to use well.

2.4 Inheritance

Object-oriented design permits class definitions to be hierarchically organized, leading to a powerful and subtle design tool called *inheritance*. With inheritance, the memory and message structures of a base class are inherited by all classes derived from the base class. (In object-oriented programming literature, the terms *base class* and *derived class* are synonyms of *superclass* and *subclass*, respectively.) Inheritance allows code to be shared among classes with a similar structure. Variations of a class can be defined by specifying in the subclass definitions only those aspects of the subclasses that differ from or elaborate on those of the base class. Messages defined in the base class may be redefined in the derived class to handle peculiarities of the derived class. The inheritance structure allows modifications to an implementation to be localized in the base class and automatically propagated to the derived classes.

Sometimes a class may be defined simply to hold structures common to several subclasses; no objects of this base class will ever be defined. Such a class is called an *abstract superclass*. Sometimes a superclass needs to be able to accept messages that must be interpreted in different ways by its subclasses. For example, if the subclasses store data of different primitive types, then different type casts will be required for each subclass. But in order for the superclass to receive the messages at all, the messages must be declared in the superclass. The *virtual function* mechanism in C++ allows the superclass to receive a message for which the interpretation must be supplied by a subclass. With virtual functions, an object may be known to be a `buffer` but whether it is actually an `int_buffer` or a `real_buffer` might be unknown at compile time. If appropriate virtual functions are defined in the `buffer` class, the selection of the appropriate code to execute will be determined at run time based on the *subclass* of the object that is receiving the message.

References

1. Rentsch, Tim, "Object-Oriented Programming", SIGPLAN Notices, vol. 17, no. 9, September 1982, pp. 51-57.
2. Stroustrup, Bjarne, "The C++ Programming Language", Addison-Wesley, 1986.
3. Brooks, F. P, "No Silver Bullet: Essence and Accidents in Software Engineering," *IEEE Computer*, vol 20, no. 4, April 1987.
4. Stroustrup, Bjarne, "What is Object-Oriented Programming?", Proc. 1st European Conference on Object-Oriented Programming, Paris, 1987.

3. Object-Oriented Programming in C++

C++ is a programming language developed by Bjarne Stroustrup and his co-workers at Bell Laboratories [1,2]. C++ is a reworking of the C programming language that provides enhanced syntactic features, eases some stylistic restrictions present in C, provides a version of strong type checking that is useful for detecting subtle errors at compile time rather than after hours of debugging, and provides new features for object-oriented programming.

This introduction to C++ assumes knowledge of UNIX and C. This introduction and the 1.0 release of COOL are based on AT&T Cfront Version 1.2.1, though the library should transfer easily to other implementations of C++.

3.1 From C to C++

C++ is an extension of the C programming language that directly supports object-oriented programming constructs and implements several extensions and improvements to C. C++ is useful as a better version of C even if one does not use the object-oriented design constructs. The high compatibility with C makes it possible for C programmers to "convert" themselves gradually to object-oriented design while remaining productive themselves and while not losing the use of the software infrastructure they have built in C. C++ still supports the high run-time speed and low-level machine control available in C, but adds tools for creating effective abstractions that help to organize large bodies of code.

The following sections are intended to provide, for C-literate programmers, a fast, concise, effective (but not comprehensive) introduction to the new features of C++. Section 3.2 describes syntactic enhancements to C that any C programmer could use to advantage immediately. Section 3.3 describes the new features in C++ that support the encapsulation aspects of object-oriented programming. Section 3.4 describes the use of inheritance in C++.

3.2 What You Already Know Probably Works

As a rule, what you know about C applies to C++.

The statement types available in C are available in C++, so much of what you see in a C++ program will be familiar. Assignments, declarations, if, for, and while statements, switch and case all work in C++ as they do in C. It is possible to write C code and compile it using your C++ processor with little or no modification. (C++ is moving toward an ANSI C foundation, and this causes many of the changes that are necessary in conventional C code to make it work under C++.) Many people take advantage of this high degree of compatibility to gradually learn and use the new features of C++ while writing mostly the C code

with which they are familiar. As they find instances where a new C++ feature is useful, they quickly learn and use the C++ enhancement. COOL is an experiment in "pure" object-oriented design, so studying the COOL code can accelerate the process of learning how to use object-oriented design.

Section 2.2 describes several enhancements to C provided in C++. These enhancements can be exploited by programmers who would like to use C++ as "a better C" but are not yet ready to dive into the new ways of thinking required for complete exploitation of object-oriented design. More significant and subtle features of C++ will be described subsequently.

3.2.1 C++ has strong type checking

The most immediate, striking aspect of using C++ is that you don't need `lint`. C++ incorporates most of the additional syntax and semantics checks that `lint` provides, without getting carried away with superfluous warning messages. Most people appreciate finding at compile time simple errors like mismatches in the number and types of arguments to procedures. Experience has shown that it is a better application of time to find and correct errors before execution begins than to wait for the inevitable but uninformative segmentation fault. (Checking of the type and number of arguments to procedures can be overridden when desired using an ellipsis ("...") in the formal parameter list as described below.)

The C++ processor will flag both assignment statements and procedure calls containing a type mismatch. The user may employ casts, but if a class is involved, there must be a constructor compatible with that particular coercion. For example, to cast a `OneType` `one` into a `TwoType` and assign its value to `TwoType` `two`:

```
/* both of these coercions are correct and legal */
    two = TwoType (one);
    two = (TwoType) one;
```

we require a constructor of the form:

```
TwoType::TwoType(OneType o)
{
    code which creates a TwoType from a OneType
}
```

Function prototypes for which checking of the number and type of arguments is impossible or not desired may be written using ellipses ("...") as follows:

```
void foo(...);           // any number and type is OK
int bar(int,int,...);    // 2 ints required, then anything
```

Note that in the these two examples the ellipses are valid C++ syntax and not an indication of something being omitted from the code example.

3.2.2 Double-slash is a comment delimiter

Comments may be initiated with a double slash. The rest of the source line is then considered a comment. It is not possible to "turn off" the comment before the end of the source line, but neither is it required to specify the location of the end of the comment. The traditional C-style comments delimited by `/* ... */` are still permitted.

Double-slash comments should be avoided in macros because they tend to interact with the C preprocessor in unfortunate ways.

```
a=5; // the rest of this line is commented out; a=10;
// the value of a is 5 - the a=10 above was ignored
```

3.2.3 Declarations may appear anywhere

C++ allows variables to be declared at any point in the code, so the variable can be declared near the code that uses it. The scope of the variable is the block in which it is defined. Thus, the following code is perfectly legal:

```
foo()
{
    int a,b;                // declare at the beginning
    a=0; b=10;
    float farray[10];       // declare in the procedure
    for (int i=0; i < 10; i++) // declare i in the statement
        farray[i]=i;
}
```

3.2.4 Variables and arguments can be declared const

Any type can be preceded by the adjective `const` which declares that the value may not be modified. `Const` is particularly useful with procedure arguments. The C++ semantics checks can prevent the disaster where a procedure modifies the value of an argument believed by the calling procedure to be a constant.

```
const int    hundred = 100;
const thousand = 1000; // if type is missing, int is assumed
const float pi = 3.14;
const char  first_letter = 'a';

// The variable i cannot be modified in the procedure below, so a
// constant may be supplied as the actual parameter.
void CannotChange(const int& i);
```

3.2.5 Parameter lists use ANSI C syntax

In C++, the formal parameter list for a procedure contains both the formal parameter name and its type. Whereas in C one would write

```
void test(a,b)
int a,b;
```

```

    {
        code for test
    }

```

in C++ the same procedure header would be written as follows:

```

void test(int a, int b)
{
    code for test
}

```

This modification (mainly due to the ANSI C function prototype syntax) is important to remember when including C header files from packages like SunWindows or X. The header files may require modifications before they can be included in C++ programs.

3.2.6 Streams provide simpler or customized I/O

In addition to the standard C I/O package, `stdio`, C++ provides an I/O encapsulation in the object-oriented style called *streams*. To use this package, your program must `#include <stream.h>` which contains the definitions of the operations in the stream package. (COOL includes this file automatically).

Stream I/O consists of classes for both input and output, called `istream` and `ostream`, respectively. Three standard streams are `cout` (for `stdout`) and `cin` (for `stdin`) and `cerr` (for `stderr`). The operators `<<` and `>>` are defined on these streams as data transfer operations and are used as follows:

```

int val;
cout << "Hello there!\n";           // write to cout

// Prompt to cout and read value from cin
cout << "Input value:"; cin >> value;

// Demonstrate writing multiple things
cout << "The number is " << value << ".\n";

```

I/O procedures for user-defined types (classes) can be defined by modifying the definitions in `stream.h` by providing overloaded `ostream::operator<<` and `istream::operator>>` procedures for the new types. (This is considered an advanced topic that will not be further discussed here.)

3.2.7 Arguments can be passed by reference

Call by reference is supported with the reference operator, `&`. C++ reference parameters are equivalent to Pascal `var` parameters. Variables passed by reference are used like any other variable and can appear on the left-hand side of an assignment. The effect of call by reference is illustrated below.

```

void refproc(int a, int& b)
{

```

```

    a=5; b=5;
}

void caller()
{
    int first=0, second=0;
    refproc(first, second);
    // now first==0 and second==5
}

```

As in Pascal, reference parameters can be used to permit a procedure to have side effects on the actual parameter (as with variable `second` above) or to prevent the copying of a large array into the procedure's environment. A parameter declared as `const int&` will be passed by reference but since the procedure promises not to change the value of the parameter by using `const` the actual parameter may be a constant in the calling procedure.

The result of a function may be returned as a reference as long as the object being returned is not part of the function's local environment (which is deallocated on the function's return). If a reference result is returned, then the function invocation may be used on the left hand side of an assignment. This can result in some valid and useful but unfamiliar constructions. For example, we might define a member function, `element` as follows:

```

class IntList{
    int list[10]
public:
    int& element(int t);
    ...
}

```

where the `element` member function returns a reference to the `t`-th integer in the `list` array. The `element` function could be used then as follows:

```

Intlist ilist;
...
ilist.element(7) = ilist.element(2)*2+ilist.element(1);

```

The function call on the left hand side of the `=` looks peculiar at first, but it is valid C++.

3.2.8 Storage management is simplified

C++ provides operators `new` and `delete` for creating and deallocating objects in the heap. `New` returns a pointer to an object of the desired class located in the heap. When the object is allocated, a constructor is invoked to initialize the object. The object remains on the heap until explicitly deallocated with the `delete` operator. `Delete` operates on a pointer to an object, invoking the destructor for the object's class and deallocating the object. The number of items to be allocated or deallocated can be specified (the default is 1). If only one object is being allocated, the constructor to use can be specified by providing an argument list to the class designator; otherwise, the null constructor (the one with no arguments) is used.

COOL works with large objects such as images and graphical models. We have developed an idiom that provides type independence in writing user code, efficient storage management, and natural, direct methods for implementing objects. The technique involves defining an "intelligent pointer class" that holds some administrative information about the object along with a pointer to an object belonging to a "storage class" that contains the mass of data. We can then manipulate the objects of the pointer class directly and naturally while the real activity is deferred to the storage class where the data is found. Type independence can be achieved if the storage class consists of an abstract superclass with subclasses for each primitive storage type. There are some rather subtle interactions between the semantics of C++ functions and the constructors and destructors of the header class (e.g. the superclass must have a virtual destructor), but when these are resolved the resulting code is elegant and efficient.

The following lines illustrate various forms of declarations with comments to indicate the meaning of each. All of the declaration below are valid C++ syntax under the given assumptions.

```
// Assume appropriate class definitions for "Point" and
// "IntList" exist.
// Here are some examples of the new and delete operators

Point p1;           // p1 is allocated on the stack.
Point* pptr;         // pptr is an uninitialized pointer;
                    // no memory is allocated for a Point.
float* fptr;         // Ditto for fptr.
IntList* ilistptr;   // Ditto for ilistptr.
pptr = new Point;    // pptr points to a new Point on the heap.
delete pptr;         // The point is deleted.
pptr = new Point[15];
                    // pptr points to an array of 15 Points,
                    // all on the heap, constructed with the
                    // null constructor Point::Point().
delete[15] pptr;     // All 15 Points are deleted.
f = new float[90];   // f is an array of 90 floats on the heap
ilist = new IntList(20);
                    // The constructor IntList(int) is invoked,
                    // and a pointer to one IntList (of size 20)
                    // is returned. The Intlist is on the heap.
ilist = new IntList[20];
                    // A pointer to an array of 20 IntLists is
                    // returned; the default (null) constructor
                    // IntList::IntList() is invoked for each of
                    // the twenty IntLists. All of the storage is
                    // on the heap.
```

C++ does not provide automatic garbage collection, so it is important to remember to delete objects allocated on the heap. It is also important to remember to specify the number of objects to be deleted if it is greater than 1, otherwise only the first object will be deleted; the rest will remain as garbage.

3.2.9 Inline functions speed execution

Small functions may be declared `inline`, meaning that the C++ processor may generate the code for the function in-line rather than coding a jump to a separate procedure. Inline functions are commonly used to simply return a value obtained from another function or to return the value of a private variable in a class. The declaration of an inline member function can take either of two forms illustrated below for member functions `f1` and `f2`.

```
class foo {
    int i;
public:
    foo();
    int f1() {i+=1; return i;}
    int f2();
};

inline int f2() // Many people prefer this form since
{              // it avoids cluttering up the class
    i+=2;      // definition.
    return i;
}
```

There are several restrictions on the complexity of computations permissible in inline functions. Furthermore, the definition of inline functions must appear in the same compilation unit as the declaration. They are usually placed in the header file for the class.

3.2.10 Overloaded functions and operators are supported

One of the most useful features of C++ for library design is operator overloading. With operator overloading, the names of procedures need not be distinct. The appropriate procedure to execute is determined by the procedure name, the number of arguments, and the types of the arguments jointly. (But note: return values are *not* used to distinguish overloaded functions.) Furthermore, symbolic operators such as `+`, `=`, `<<`, `+=`, `*=`, etc. may be redefined to have valid interpretations for arguments of any class.

Operator overloading has important uses in COOL. COOL contains some code that is type-independent; we do not need different procedure names for performing the same operation on different kinds of objects. Thus, when using COOL matrices, vectors, and complex numbers, we may use the natural notation `*` or `*=` for multiplication; we do not need to remember whether matrix multiplication is `mpy` or `mult` since the operators `*` and `*=` are defined appropriately for each class. Also, we do not have to remember different procedure names for multiplying a matrix by a scalar, a vector, or a matrix; all these operations use the `*` or `*=` operator and return an object of the appropriate type.

Operator overloading permits the user to employ more natural notations and to be less concerned with incidental matters like the storage types of values being operated upon.

The following class has an overloaded constructor, an overloaded member function, and an overloaded operator:

```
class MathThing {  
    ...  
public:  
    MathThing(int);    // Construct a math thing from an int.  
    MathThing(float); // Construct a math thing from a float.  
    double MyExponent(int);    // Overload the member fcn  
    double MyExponent(MathThing); // MyExponent.  
    MathThing& operator=(MathThing& x); // Overload the =  
}
```

To overload a function that is not a member function of a class, you must first specify

```
    overload functionname;
```

and then overload the function:

```
void functionname (formal parameter list)  
{ .... }  
void functionname (a different formal parameter list)  
{ .... }
```

Operator overloading can yield particularly elegant code when used carefully. For example, the statement

```
int i, j;  
i = i * j;
```

clearly refers to multiplication of integers. The statements

```
Matrix m1, m2;  
m1 = m1*m2;
```

are natural but are undefined in C++. By defining a `Matrix` class and definitions for the multiplication and assignment of matrices, the above statements can be made valid and meaningful.

The indexing operator `[]` is often overloaded in order to include bounds checking. A complete definition of an overloaded operator `[]` for an integer array class called `IntArray` is given below. The example assumes that `BADVALUE`, `size`, and `buffer` are defined in the class declaration.

```
int IntArray::operator[](int index)  
{
```

```

        if (index >= size)
        {
            cerr << "array index out of bounds\n";
            return BADVALUE;
        }
        return buffer[index];
    }

```

The first line indicates that this is an `operator[]` member function of class `IntArray` that returns an `int` result and has one `int` argument named `index`.

The same precedence rules apply to overloaded operators as to standard operators. Precedence rules in C++ are the same as in C.

The following operators can be overloaded:

| | | | | | | | | |
|--------------------|---------------------|-------------------------|-----------------------|-----------------------|------------------------|------------------------|-----------------|-----------------|
| <code>new</code> | <code>delete</code> | <code>-></code> | | | | | | |
| <code>+</code> | <code>-</code> | <code>*</code> | <code>/</code> | <code>%</code> | <code>^</code> | <code>&</code> | <code> </code> | <code>~</code> |
| <code>!</code> | <code>=</code> | <code><</code> | <code>></code> | <code>+=</code> | <code>-=</code> | <code>*=</code> | <code>/=</code> | <code>%=</code> |
| <code>^=</code> | <code>&=</code> | <code> =</code> | <code><<</code> | <code>>></code> | <code>>>=</code> | <code><<=</code> | <code>==</code> | <code>!=</code> |
| <code><=</code> | <code>>=</code> | <code>&&</code> | <code> </code> | <code>++</code> | <code>--</code> | <code>()</code> | <code>[]</code> | |

3.3 C++ Support for Encapsulation

This section will describe the language features available in C++ to support the encapsulation aspects of the object-oriented programming methodology. A discussion of the C++ features that support the inheritance aspect of object-oriented design and the usage of inheritance will follow in the next section.

The example we will develop in this section illustrates the fundamentals of defining and using classes. This example defines a binary tree node (`btnode`) having left and right subtree pointers and an integer data value. Integers are inserted into the tree so that an infix traversal of the tree visits the data values in increasing order. This example is not concerned with balancing the sorted tree.

3.3.1 Classes Define the Structure and Function of Objects

A class is a collection of data structures and operations on those data structures. The class definition is just a compiler declaration (like a user-defined type declaration in Pascal or a `struct` definition in C); no storage is allocated or other operations performed until an instance of the class (an object) is created.

The general structure for a class definition (without inheritance) is as follows:

```

class classname
{
private:
    private data and member functions
    (the keyword private: is the default and is
     usually omitted)

```

```

    public:
        public data and member functions
    };

```

The syntax for defining a class is illustrated through the following sections with the complete definition of a node suitable for constructing a sorted binary tree of integers. The class declaration is as follows:

```

class btnode
{
    // Private stuff comes first
    btnode* left;
    btnode* right;
    int data;
    btnode& insert(int); // note: a private member function

public:
    // User interface routines follow

    btnode(); // "null" or "default" constructor
    btnode(int); // another constructor
    ~btnode(); // destructor
    btnode& print_infix();
    btnode& operator+(int); // These messages illustrate
    btnode& operator+(btnode); // operator overloading.
};

```

After the name of the class, several private data items and one private procedure, `insert`, are declared. The public part of the class definition begins with the keyword `public:` and lists the operations available to users of `btnode` objects including the constructors, the destructor, and three other member functions.

Objects of this class are created by declaring them using the class name in the role of a type or by explicit allocation on the heap using the `new` operator.

3.3.2 Constructors and Destructors Create and Destroy Objects

Constructors and destructors are invoked automatically and return no value since their purpose is to create side effects in the object being constructed or destroyed. Proper use of constructors ensures that all objects of the class are given a valid, consistent state when they are created. Proper use of the destructor ensures that all non-local memory allocations made by the object are cleaned up before the object is deallocated, thereby preventing accumulation in the heap of garbage data from deleted objects.

Constructors have the same name as the class; the two procedures called `btnode` are constructors in the example. A constructor is invoked to initialize any object of this class when it is created. The default constructor is used unless the declaration of the new object specifies arguments that select another constructor.

For example, a `btnode` may be created in a user program in the following ways:

```

btnode Zeronode; // Zeronode is an object of class btnode

```

```

// initialized with the default constructor.
btnode One(1); // One is a btnode initialized by the
               // constructor btnode::btnode(int).

btnode* btptr;
btptr = new btnode; // This btnode is allocated on the heap
                   // using the default constructor
                   // Remember: new returns a pointer!

```

The code of the two constructors for btnode follows:

```

btnode::btnode()
// the null constructor initializes the
// object to all default values
{
    left=NULL;
    right=NULL;
    data=0;
}

btnode::btnode(int value)
// This constructor initializes the object
// with a particular data value as
// specified by the argument "value"
{
    left=NULL;
    right=NULL;
    data=value;
}

```

The name of the destructor is a tilde followed by the class name; in the example, the destructor is called `~btnode`. The destructor is invoked automatically whenever an object of the class is destroyed either by explicit deallocation using the delete operator or by automatic deletion when the object goes "out of scope" (at the termination of a procedure for which the object is a local variable).

In the example, the destructor must ensure that when the current `btnode` object is deleted that the `btnode` objects to which it points also receive the delete message. Without this action, the current node would be deleted but its subtrees would remain in the heap as garbage. The code of the destructor `~btnode` is as follows:

```

btnode::~~btnode()
// the destructor has to be sure that the subtrees
// get deallocated properly
{
    if(left<>NULL) delete left;
    if(right<>NULL) delete right;
}

```

Each `btnode` will invoke its destructor, sending the delete messages throughout the tree. (Actually there is only one copy of the destructor code that is shared by all objects of the class.)

Most constructors and destructors are short, simple procedures. Classes that do not allocate any space on the heap often have destructors with null bodies, written `{}`. Constructors and destructors may be inline procedures. If a class is complicated, requiring a more involved initialization sequence, the initialization code may be shared by all of the constructors by placing most of the initialization code in a private procedure (named `init()` or `make_a_btnode()`, for instance) that is invoked by the various constructors.

3.3.3 Private Data and Procedures Define Internal Structure

The declarations after the class name and before the `public:` keyword define private components of the class. (There is also a `private:` keyword that is rarely used since the default access at the beginning of a class declaration is `private:.`) Private data and procedures cannot be accessed by the procedure that declares the object or by the member functions of other classes.

In the `btnode` example, the two pointers `left` and `right` and the integer data are private, as is the procedure `insert`. A tenet of the object-oriented programming convention requires that users of an object know nothing of its internal structure, so data members of a class are kept private. A user seeking the value of a data item must request the value through a member function. This rule has a foundation in software engineering practice: the data value could later be moved to another class, so by accessing the value through a member function we permit modifications in how the value is obtained without requiring modifications in the users of the class. Private procedures are usually either utilities needed by other methods or a procedure defined for use in the another method that would clutter the code there. The `insert` procedure is an example of both rationales, though in this simplified example it is invoked only once in another member function.

The code of procedure `insert` is shown below. It enters a given integer value into the sorted tree of `btnodes`. The `insert` procedure has access only to the current node, so if the integer is not to be attached to the current node it passes the integer down to be inserted in one of the subtrees. Note that this is, in effect, a recursive call from `insert` to itself, though the object being asked to invoke `insert` is a different `btnode`. When reading a member function definition, remember that the member function has access to the private variables of the object that is invoking the function as they were defined in the class definition. Thus, `data`, `left`, and `right` are defined already and refer to the values for the object that is executing the function. In addition, every member function has access to `this`, which is a pointer to the object that invoked the function. I prefer to have access to the object directly rather than through a pointer, so I always include

```
#define self (*this)
```

which defines `self` as the object to which the `this` pointer points. I can therefore return the object as the value of the function, as illustrated below. The

insert function also illustrates the syntax for sending a message to an object through a pointer by using the `->` operator.

```
btnode& btnode::insert(int i)
{
    if (i<data) then          // is i less than this node's value?
        if (left=NULL) then   // is there a left subtree?
            left=new btnode(i); // allocate a new btnode
                                // with value i
        else
            left->insert(i);    // left is a pointer;
                                // use -> syntax
    else if (i>data) then
        if (right=NULL) then  // is there a right subtree?
            right=new btnode(i); // allocate a new btnode
                                // with value i
        else
            right->insert(i);   // send to right subtree
    return self;
};
```

3.3.4 Public Member Functions Define Externally Visible Structure

The `btnode` class provides several operations for use by any procedure or other object. The code for these procedures is shown in full below.

The `print_infix()` procedure prints to cout the contents of the entire tree whose root is `self` using an inorder traversal. Thus, if the `insert` procedure works as expected, the values in the tree will be printed in ascending order.

The other public member functions redefine the `+` operator for use with `btnode` objects, providing a simple, natural syntax for adding new objects to the tree.

Member functions of a class can be invoked on an object in two ways. To invoke the member functions on the object itself (rather than invoking the function on the object indirectly through a pointer) the syntax is

```
object.func(args);
```

To invoke a member function on an object that you access through a pointer, the syntax is

```
objectptr->func(args);
```

In the former case, the variable `object` was declared as an object of its class. In the latter case, the variable `objectptr` was declared as a pointer to an object of the appropriate class (and the actual object was probably created on the heap by the `new` operator). Confusing the semantics of your program and using the wrong invocation mechanism will result in syntax errors.

If member functions return references to `self` when they would otherwise return `void`, it becomes possible to chain messages together as follows:

```
object.dothis(args).dothat(moreargs).dotheother(evenmoreargs);
```

The messages are handled in order, left to right.

Several points about these member functions are worthy of note. First, each of them returns a reference to a `btnode`. In my idiom, a member function that returns a reference to its own class is returning `self`. This makes possible the chaining of operations on a single object. It is reasonable to return `self` from any procedure that would otherwise return `void`. Second, notice that two of these functions are named `operator+`. They are distinguished by the types of their arguments. One adds an integer to the tree, the other adds a tree to the current tree. Third, note how the `operator+` functions send messages to `self` (`self.insert(i)`; or `self+b.data`; for example) in order to modify the current object. Sending messages to `self` is particularly important in combination with inheritance since a message to `self` is interpreted through the entire inheritance hierarchy and not just with respect to the current class. It is therefore possible to invoke functions defined in superclasses by sending the appropriate message to `self`.

```
btnode& btnode::print_infix()
// This message uses recursive calls to itself to print the
// tree in infix (sorted) order
{
    if(left!=NULL) left->print_infix();    // print my left
    cout << data << " ";                // print my data
    if(right!=NULL) right->print_infix(); // print my right
    return self;
}

btnode& btnode::operator+(int i)
// add an integer to the sort tree
{
    self.insert(i); // (note: sending a message to myself!)
                    // since self is an object, use "." syntax
    return self;
}

btnode& btnode::operator+(btnode b)
// add a tree into self!!
{
    if(b.left<>NULL)then
        self+(*b.left);    // * dereferences the pointer
    if(b.right<>NULL) then  // note the recursive calls to +
        self+(*b.right);
    self+b.data;           // This one is NOT recursive!
    return self;
}
```

3.3.5 Programs use classes as if they were predefined types

The code and output of a `main()` program using class `btnode` is shown below. The program begins by using `#include` directives to read the definitions of the C++ stream classes and the `btnode` class. We assume for now that the `btnode` class definition is in file `btnode.h` and the definitions of its member functions are all stored in file `btnode.c`.

```

#include <stream.h>
#include "bnode.h"
#include "bnode.c"

main()
{
    bnode s,t;           // define 2 default bnode objects
    s+6+3+2+5+8+7;      // chaining is possible because each
    t+4+9+1+10;          // function returns self.
    s.print_infix(); cout << "\n";
    t.print_infix(); cout << "\n";
    s+t;                 // tree plus tree gives tree!
    s.print_infix(); cout << "\n";
}

```

Output from main():

```

0 2 3 5 6 7 8
0 1 4 9 10
0 1 2 3 4 5 6 7 8 9 10

```

3.4 Using Inheritance

Inheritance is a subtle and powerful technique whereby a new class (called the derived class or the subclass) can be declared as being derived from an existing class (called the base class or the superclass) thereby inheriting all of the data and operations of the base class. Inheritance is used to factor out commonalities among a group of classes and then to implement common routines or common portions of routines once in the base class. Alternately, the relationship may be viewed as a subclass providing refinements or customizations to the definition of the superclass. However you think about it, the code written for the superclass is shared by all of the derived classes. Member functions of derived classes may invoke superclass functions from any level up the inheritance hierarchy, modify or replace superclass functions, or even hide them.

Several features have been added to C++ to support inheritance, and they are described in this chapter.

3.4.1 Declaring a subclass

To declare a subclass, we make a notation in the class declaration as follows:

```

class derived1 : public base{...}
class derived2 : base{...}

```

In the first declaration, which is most commonly used, the public members of class base are also public in class derived1. In the second declaration, the members of base are private in class derived2. Thus, the member functions of derived2 can invoke member functions of base, but users of derived2 cannot access the members of derived2 that are defined in base.

3.4.2 Accessing Members of the Superclass

An object belonging to a derived class contains all of the members declared in both its own class declaration and in the superclass declaration(s). Members of the superclass may be accessible to the derived class object or they might be restricted based on the access permitted in the superclass declaration as described below.

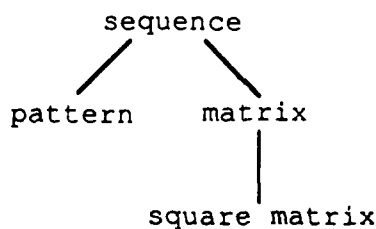
Three types of access can be specified in a class definition. Private members are accessible only to objects of the class being defined. Public members are accessible to objects of any class or to ordinary program segments. Protected members are accessible only to objects of the same class or to objects of its derived classes.

Typically, the private part of a class declaration includes all of the data members and a few utility functions used in constructing the class. For example, a complicated class requiring significant initialization work might have a dozen constructors, meaning that there are a dozen different ways by which objects of this class might be specified. If the initialization effort is similar for all of these construction methods, then the initialization code might be placed in a private utility function that is called by the constructors, perhaps after some preprocessing of their arguments to rearrange them into a common form.

The protected part of a class consists of functions that subclasses might use but that are of little interest to users of the class. These are usually utilities that are shared among the subclasses.

3.4.3 The Semantics of Inheritance

We begin this discussion of inheritance with an example of an inheritance hierarchy that will be used to illustrate the concepts. We will develop an inheritance hierarchy for a sequence class that contains an array of floats and then derive from it classes for pattern, a 1-D sequence, and matrix, a 2-D sequence. Then from matrix we will derive a square_matrix class. The inheritance hierarchy is diagrammed below. (This hierarchy is based on part of the COOL nseqlib sublibrary.)



Each subclass refines or specializes its superclass. This refinement can involve new data (a matrix must contain its row and column dimensions, which are not required for a sequence which needs only its length) or new operations (inverse and determinant are defined only for square matrices).

```

class sequence
//
// This class defines a sequence of floats with only basic
// operations defined that are valid for both 1D and 2D sequences
{
    float* data;
    int seq_len;
protected:
    sequence();
    sequence(sequence&);
    sequence(int);
public:
    sequence& zero(); // set all elements to 0.0
    sequence& copy(sequence&); // copy given sequence to self
    sequence& replace(sequence&); // adopt size and content of arg
    sequence& add(sequence&); // add element by element
    sequence& sub(sequence&); // subtract element by element
    sequence& scale(float); // multiply each element by arg
    sequence& operator=(sequence&); // this does a replace
    float& operator[](int); // get a particular element
};

class matrix : public sequence
{
    int rowsize, colsize; // A matrix has extra data
public:
    matrix();
    matrix(matrix&);
    matrix(int rows, int cols); // A new constructor type
    matrix multiply(matrix&); // A new operation
    matrix& multiply(float); // Scalar multiplication
    matrix& operator=(matrix&);
    matrix operator*(matrix&); // Another version of multiply
    matrix& operator*=(matrix&);
    float& operator()(int, int); // A 2-D subscripting operator
};

class square_matrix : public matrix
{
public:
    square_matrix();
    square_matrix(square_matrix&);
    square_matrix(int dimension); // only 1 dimension needed
    square_matrix inverse(); // new operations
    float determinant();
    square_matrix& operator=(square_matrix&);
    square_matrix operator*(square_matrix&);
    square_matrix& operator*=(square_matrix&);
    square_matrix operator~(); // matrix inverse
    square_matrix operator!(); // matrix transpose operator
};

```

When an object of a derived class is created, a constructor of the superclass is invoked first, then the specified derived class constructor is invoked. Thus, the derived class constructor can assume that its inherited components are defined from the superclass before it begins operation. Sometimes, most of the initialization can be undertaken by the superclass constructor if the correct one can be specified from the subclass. A special syntax is provided to do just that. Class `matrix` has a constructor of the form

```
matrix::matrix(int rows, int cols)
```

The work of this constructor can be performed mainly by the sequence constructor

```
sequence::sequence(int length)
```

by using the following syntax in the definition the `matrix` constructor, which is completely specified below:

```
matrix::matrix(int rows, int cols):(rows*cols)
{ rowsize=rows; colsize=cols;}
```

The expression in parentheses after the colon specifies the arguments to be provided to the superclass constructor. In this case, the sequence constructor expects an `int` and it gets an `int` formed by the product of the arguments of the `matrix` constructor. The sequence constructor is executed first, then the `matrix` constructor may begin its work assuming that the specified sequence constructor has executed. All that is left for the `matrix` constructor is to record the row and column sizes in the private variables `rowsize` and `colsize`.

This technique of passing the work up to the superclass is an important feature of inheritance that tends to cause maturing systems to shrink as more commonalities are noticed by the designer and passed up the hierarchy to successive superclasses. An example of this is how the `matrix::multiply(float)` member function can invoke `sequence::scale(float)` to perform its work. Here is a complete definition of the class `matrix` member function `multiply`:

```
matrix& matrix::multiply(float)
{
    self.sequence::scale(float);
    return self;
}
```

Object-oriented programming yields programs that appear to work like an effective bureaucracy: objects shuffle messages off to somewhere else for processing; the results return, are stamped, and are returned to the sender.

By prepending the class name and double colons as in the previous example, a particular member function of a superclass may be invoked, even if the name has been overloaded in derived classes or even in the current class. Thus, in a

member function of `square_matrix`, one might invoke a member function of sequence by writing:

```
self.sequence::add(sm);
```

Access to system functions can be obtained by prepending just the double colon. Thus, if `cos` has been overloaded, the system's `cos` function can be obtained by invoking `::cos`.

To make inherited operations more easily usable by users of the derived class, it is common to provide a stub specific to the derived class that simply calls the superclass procedure. This is especially true for operators since they must be treated as functions and cannot work just by side-effects. Thus, in our example, there is a `square_matrix::operator*(square_matrix&)` member function that simply calls the `matrix::operator*(matrix&)` operator. This makes using the operator easier for `square_matrix` users since if the member function from `matrix` were used directly its result would be a `matrix`, requiring a cast to assign the result to a `square_matrix` variable.

Subscripting operators can be conveniently defined to return a reference to the array element, thereby allowing assignment to the result of the function. Thus, natural usages are possible such as:

```
x(5,3)=x(3,5)*2.0;
```

3.4.4 Member Functions Most Classes Need

There are several member functions that almost every class needs for reasons of administrative necessity or ease of use. These are:

A default (null) constructor. The default constructor is used to initialize each element of an array of objects of a class. Its purpose is to ensure that an object of the class can be created on demand even if no useful argument sequence is known or can be specified. If the creation of an object requires certain external conditions to hold, such as a disk file constructor requiring the file name not to already exist, then a null constructor might not be provided, thereby requiring the user to specify parameters, a file name, perhaps, and preventing the user from creating an array of objects of the class.

A constructor that takes a reference to an object of the class as its argument (a copy constructor). This constructor can be invoked by the user to create an object, though more often the simultaneous declaration and assignment will be preferred, as in

```
sequence s = sequence(8);
```

More important, the copy constructor is invoked automatically when an object of the class is to be returned as the value of a function. Its purpose in that situation is to copy the return value from the function's scope to the calling procedure's scope.

A destructor. The destructor is necessary if the constructor allocates any variables or objects using the new operator.

An assignment operator. Since assignment is such a common operation, you might as well go ahead and define it now because you will not feel like it later.

Access routines. In the culture of object-oriented design, manipulation of an object's private data is considered impolite. Access functions return the values of private variables of the object, making direct access to the object's data unnecessary. This approach is desirable since the internal structure of the object could change, and the extent of changes to calling procedures is minimized by invoking a function rather than by grabbing the data values directly. For example, the data items could be migrated to another class. Rather than rewriting the user programs, the access function could simply ask the object of the other class for the value rather than returning the value from itself. This internal design change would have no effect at all on the user procedures.

3.4.5 Virtual Functions

One of the most hideously error-prone constructs in C is the pointer-to-a-function. Even worse are transfer vectors, which are arrays-of-pointers-to-functions. One of the great contributions of C++ is to make it possible to hide these messy constructs in a compiler-checkable syntactic and semantic feature called virtual functions.

The concept and operation of virtual functions is not simple, but it can be very powerful - as powerful as transfer vectors. The notion is as follows:

Suppose you have a pointer to an object of a base class like `sequence`. You are manipulating an object that you know is a `sequence` or some derivative of `sequence`, and you really don't care which. What you would like to do is to ask this object to do something - something that all `sequence` objects should know about - and be able to expect a reasonable response. Now the action that the object actually takes may depend on which derived class the object belongs. For example, if you tell a `buffer` object to add 5.75 to each of its elements, how this action is carried out depends on the storage type of the elements in the buffer. If the storage type is `float`, you expect different results from the case where the storage type is `int`. If `buffer::add(float)` is declared virtual, then the `add` message may be sent through a `buffer*` variable to an object and the C++ run-time system will select the appropriate procedure to execute based on which subclass of `buffer` the object really is: `int_buffer` or `float_buffer`, for example. This technique is used in COOL to provide type independence in the `bufferlib` sublibrary, which in turn, allows for type-independent user manipulation of image objects.

Another example of virtual functions in COOL occurs in the `graphlib` sublibrary where a linked list of graphical object is being created. There are two subclasses of class `glob` called `gobject` and `group`. When a graphical model is being constructed, one could attach any of these classes to any other one. Since the types of the objects cannot be determined at compile time, I defined several virtual functions of `glob` that are implemented differently in objects of the subclasses. The additions to the base class that make the subclasses unique and different

from the base class require different processing in certain cases, so the use of virtual functions makes possible the selection of the correct procedure to execute at run time.

Virtual functions are implemented in C++ by a virtual function table (vtbl in error messages). The virtual function table is basically a transfer vector in which the entries defined by the base class are overwritten by the pointers to virtual function implementations from the subclass. The result is a powerful run-time operation provided by a simple syntactic addition that hides some of the most error-prone and difficult-to-debug code that is available to write in C.

References

1. Lippman, S. B., *C++ Primer*, Addison-Wesley, 1989.
2. Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, 1986.
3. Pohl, Ira, *C++ for C Programmers*, Benjamin Cummings, 1989.

4. Designing C++ Libraries

4.1 The Challenge of Software Library Design

Design is the art of recognizing, evaluating, and selecting tradeoffs. In software development, design involves selecting from the multitude of expressions available those particular forms that result in "good" code - code whose structure, function, and operation optimize certain criteria. Software design is difficult: the designer must weigh arbitrary, interacting constraints and conflicting objectives. This difficulty is essential to the enterprise, not an incidental, temporary lack of technological development, but the incidental difficulties are not negligible [1]. Tools, languages, or techniques that ease the incidental difficulties of expressing, understanding, and updating ideas embodied in software are badly needed. Object-oriented design provides such relief by specifying some criteria concerning methods of encapsulation and a semantics of inheritance that have been incorporated into several new programming languages, including C++ [7, 13].

To say that a library is written using object-oriented design says something about the mechanisms used to code and package the library in a programming language but nothing about the criteria used to select and evaluate the tradeoffs that form the library. Library writers and their clients must understand that an "object-oriented library" does not mean a "well-designed library"; object-oriented design is not a sufficiently comprehensive or detailed discipline to support such a claim [11,14].

Moreover, C++ leaves the programmer the maximum flexibility of expression - a language design criterion inherited from C [13]. This is good in that there are relatively few arbitrary restrictions and many alternative strategies by which a programmer can express an elegant idea; in fact there are so many ways to express almost any idea that there is certainly an elegant expression in there somewhere. The flexibility of C++ is bad in that inexperienced (and often, not-so-inexperienced) programmers are too often tempted to select poor tradeoffs because they lack understanding of the multitude of options that are available.

Regrettably, the ignorance on the part of computer programmers concerning good design is compounded by the lack of a culture in which programmers study good designs and communicate design ideas to others for review [16]. We tend to want to learn from experience rather than from study. We are ready to adopt anything that works, especially if it can be obtained free. The growing array of options available through object-oriented programming and personal workstations with color bit-mapped displays (a fertile application field for object-oriented design) overwhelms the ability of psychology to provide scientific data on how this new programming and user interface technology might be used effectively, reliably, and safely.

The result is that there is some truly ridiculous code being written out there; all of it, however, written in one of the popular new languages, and all of it adhering to

(some form of) the Object-Oriented Programming Discipline (with capitalization as if this Discipline were carved in the side of a mountain for everyone to visit and admire - in fact, OOP is not a discipline at all, but a convention.) [11]. Fortunately, I have been able to trace every ridiculous implementation question I have been asked to the ridiculous design concept that caused it. If we were better designers, we would know that *encountering the need for a ridiculous hack during implementation is an indictment of our design, not a challenge to our hacking skills.*

Because library designers have so many options and so little experience with them both individually and collectively (even if we were in the habit of using our collective knowledge effectively), we need to specify, evaluate, and select specific criteria that will govern how we evaluate the tradeoffs we face in the design of object-oriented libraries. Informed consumers of such libraries should demand of the library supplier some discussion not only of what is in the library and how it works but also of what criteria were used to select what is in the library and why it is expressed as it is.

C++ is a new programming language advocating new software design methods and programming language technologies, so it is not surprising that there has been considerable discussion concerning how the language and the methods it supports can be employed most effectively. This paper will review three design strategies used, even advocated, for C++ programs and libraries, discuss serious problems with all three approaches, and propose an alternative set of design criteria, more specific than the three common approaches, that has proven effective in the author's C++ library.

Three design strategies current in the practice and literature of object-oriented software design are

- (1) comprehensive, monolithic class hierarchies,
- (2) collections of tiny encapsulated tools, and
- (3) flow control residing in the user interface code.

4.2 Why comprehensive, monolithic hierarchies are a bad idea

The notion of defining a single hierarchy containing all of the classes one might ever need was implemented in the Smalltalk system [5]. Every class is a part of a single hierarchy with object as its root. The Smalltalk design has been transplanted into other object-oriented languages such as Objective-C's ICpack 201 [12] and even C++. The NIH class library (formerly called OOPS) is a transliteration of a large subset of the Smalltalk hierarchy into C++ [6].

Smalltalk is an interpreted system that gives up a great deal of performance in order to achieve an amazing degree of flexibility. Since the definition of the Smalltalk system is accessible to the programmer, each programmer can create customized but incompatible Smalltalk environments. The main reasons for using the more conventional object-oriented languages rather than Smalltalk is that performance, portability, and compatibility matter. The tradeoff one accepts in selecting a more conventional language is that changing one's environment is

more difficult than in Smalltalk. A monolithic class hierarchy is less troublesome in Smalltalk than in other languages because the Smalltalk environment can be changed so easily. More conventional operating system and language environments are much more difficult to change. Thus, *imposing a monolithic hierarchy locks users into a single implementation of each class that is difficult to change.*

The monolithic hierarchy is particularly inappropriate in C++. Implementation of a monolithic hierarchy requires extensive use of macros to fake polymorphism and complicated class names to encode type information in order to satisfy the strong type checking of C++ [6]. The result is object-oriented code that is more error-prone, harder to read and understand, with even more memory burden on the programmer due to even more complicated names to remember. Rather than helping programmers to control programs, monolithic hierarchies combine with design decisions implemented in C++ in unexpectedly nasty ways to produce more of a mess than we faced with conventional C. *Name spaces make poor databases.*

Moreover, the use of a monolithic class hierarchy is contrary to the design criteria of C++. C++ was designed to provide as many implementation options as possible within its constraints, not to dictate solutions or solution methods [13,14]. To use C++ in a way that restricts those options in costly, ineffective ways is ironic.

4.3 Why a toolkit of tiny classes is a bad idea

The worst aspect of object-oriented design is that it tends to encourage bottom-up design instead of top-down design [14]. Top-down design (also called Stepwise Refinement [17]) is a heuristic that suggests that better software systems result if the application-level concepts guide the design process rather than the low-level implementation structures that happen to be available or convenient [2,4,17]. The tendency to favor the bottom-up approach while using object-oriented design causes the greatest disasters when it leads programmers to design libraries of tiny classes from which, supposedly, larger classes may be constructed. The fallacy of this approach is easy to see from several viewpoints.

From an economic viewpoint, to buy a class (that is, to obtain, learn about, and use somebody else's class definition and code) saves the buyer certain costs in developing and debugging the code while exacting certain other costs: First, the author's approach must be understood and accepted, including its impact on the buyer's subsequent design decisions. Second, the buyer must relinquish a measure of control over the product; the buyer is now dependent to some extent for bug fixes and enhancements on the author of the library. The economic point of view says that the value of the library must exceed its costs or it will not be bought. From this viewpoint, building collections of tiny classes does not maximize the likelihood that other people will buy the library: *Small encapsulations are small benefit.*

From a user viewpoint, also, the toolkit of tiny classes is not the most desirable way to obtain labor-saving software. The user has a problem to solve, and the

solution requires code to be written. What the user wants is an encapsulation of the concepts required to solve the problem, not a set of polymorphic stacks and queues. The most valuable kind of software from the user's viewpoint is a tool that matches the need, not a collection of low-level implementation hacks to which the needs must be retrofitted. *Tiny classes are not what the user wants.*

From a software-engineering viewpoint, a toolkit of tiny classes employs object-oriented design in a way that fails to achieve one of the main rationales for adopting object-oriented design: organizing the name space. The library of tiny classes fails to decrease the surface area of the library (the number of names accessible at some point in the code [3]) and, in the context of strong type checking, complicates the names that are used. A name space makes a poor database. This is exactly why conventional libraries with flat name spaces are difficult to use as the library gets large. A library of tiny classes encounters the same problem, only slightly later because the small encapsulations that are used do contribute a little to the organization of the name space. *Building a library of tiny classes is a bad design strategy.*

This discussion in particular has important implications for teaching object-oriented design. For one thing, it implies that the common example of inheritance where rectangle and circle are derived classes from polygon reflects a bad design strategy because the classes are too small to be of any benefit.

4.4 Why UI-controlled implementation is probably not a great idea

Computer people are a friendly, cooperative folk until we enter the arena of user interface design. On this topic, programmers are passionate, interested, parochial, bigoted, and deeply ignorant. And the situation is not getting better. The appeal of user interface design is clear: the immediate, visible results and the unlimited potential for midnight hacking are powerful aphrodisiacs in this community. In addition, a user interface package works with ideas that fit naturally into an object-oriented design methodology, so object-oriented user interface work is aesthetically satisfying.

Color bit-mapped displays have opened new worlds for user interface designers to explore. The complex software systems that control these displays consume many compute cycles and require much attention from the application software; so much attention that many people are advocating that the operation of application programs be controlled by the user interface. That is, after initialization (and the required initialization is extensive), control is given to the user interface system which invokes application procedures via "callback functions" as required by the user interaction [8,15].

This design strategy gives a deserved importance to the user interface and facilitates highly responsive user interaction since nothing happens until the user commands it; the application is interrupt-driven based on the user's actions. It has been painful to me to find flaws in this approach, but the main flaw is less serious than with the other two design strategies.

There are several tradeoffs in using the UI-controlled design strategy. The user interface control system must be extremely complex in order to handle the variety of interactions that might be required. The structure of the desired user interface must be communicated to the user interface control system, of course, but in addition, the structure of the *application program* must be communicated to the user interface through callback functions. Thus, the initialization of the user interface system is extremely complex, error-prone and tedious.

The problem is that the user-interface control strategy does not map gracefully into an object-oriented design. The user interface is not an "object" any more, and it is not separated from other aspects of the application. The structure of the application must be defined to the user interface, adding a burden of messy, error-prone procedure calls atop the application. In effect, the application must be written and then its structure must be "explained" via the user interface definition and the callback functions to the user interface control system. While the user interface control system might be written in an object-oriented manner, its appearance to its client applications is like that of kudzu: intertwined, going where it shouldn't, and next to impossible to manage neatly. *UI-controlled implementations are not object-oriented implementations.*

4.5 Objectives of Library Design

The purpose of a library is to encapsulate useful implementations in a form that is easy to use. The principal design objective of a library architect is to create and exploit a separation of concerns [9,10] between the library user and the library implementer. This separation of concerns is achieved by defining encapsulations that are useful and that are, to the extent possible, self-contained. The architect endeavors to minimize instances where to use a module in the library it is necessary to know about the internal structure of that module or about the structure or operation of other modules in the library.

Object-oriented programming languages contribute to library design by supporting the separation of concerns created by the library architect. However, this does not mean that an object-oriented library is necessarily a well-designed library. Some further guidance is necessary to improve the chances of obtaining a well-designed library.

I have found that there are three different kinds of concerns that face library developers and users. First are the concerns of **processing**, including how data items might be teased apart to form a more complex internal structure to facilitate processing and what process encapsulations are required to get the job done. Second are the concerns of **interaction and display** where one encounters many hardware dependencies and where timing of interrupts and input sampling is likely to be critical. Third are **storage and communication** concerns where formatting of bytes in a standard sequential order is an essential problem. I submit that designing class libraries to separate these concerns leads to elegant, effective, and useful encapsulations.

4.6 Library Design in Practice

This new design criterion immediately affects how one selects and implements encapsulations in a library. Three examples from my library, COOL, which is concerned with image processing, image pattern recognition, and computer graphics, illustrate the impact of the criterion in practice.

The image class is central to my library. I originally wrote the image class with `load()` and `save()` member functions so that an image knew how to load itself from disk and save itself there. These were the longest member functions in the whole image class, so they already looked out of place there. When I later found that I needed to `load()` and `save()` certain matrices, I realized that the code I had developed to implement loads and stores of images could not be used (via inheritance) to help me define analogous operations on matrices. This struck me immediately as a serious design flaw. I decided to bring together all disk handling routines under a single inheritance structure in which basic disk file services are provided by a base class, `diskfile`, which conveniently encapsulates UNIX file handling procedures, while file formatting information for different kinds of files is hidden in subclasses of class `diskfile`.

The representation of an image on disk is encapsulated by class `imagefile`. To read an image from disk one creates an `imagefile` with appropriate parameters and then invokes

```
image imagefile::load(int plane=0).
```

The image returned will be a plane from the `imagefile` (which can hold many image planes). Similarly, to save an image to disk, one invokes

```
imagefile& imagefile::save(image im, int plane=0).
```

The `imagefile` object does not need to know the internal structure of an image - it does need to know how to ask an image for whatever components of the image are required to create a disk representation of it.

Another example occurs in my `imagetool` and `Ximagetool` classes, which encapsulate image display on workstations under SunView [15] and the X Window System [8], respectively. To display an image on a workstation running X, one creates an `Ximagetool` object and invokes the member function

```
Ximagetool& Ximagetool::display(int viewport, image im).
```

The `viewport` argument specifies the quadrant of the `Ximagetool` window in which the image will be displayed.

The real contribution of the `imagetool` and `Ximagetool` classes is to abstract the essential qualities I want in an image display, on whatever device the display appears. All three of my image display classes (the third is called `ikonas`) have identical public interfaces, but their internal structures are very different. My library design criterion requires that display and interaction be separated from processing concerns, so the display of images is encapsulated and separated from the code that is concerned with the processing of images. By encapsulating the code that is dependent on the display system, I know exactly what needs to be rewritten to enable my style of image display on another device or under another user interface control system. The port of `imagetool` to X took just a few hours,

and there was never any doubt that everything that needed to be ported was indeed ported.

Now we will consider a possible future addition to COOL that appears at first to raise problems for my library design criterion. Suppose we want a class, `net_display`, that can display an image either on the local workstation or on a remote workstation in a manner that is transparent to the user. This class must involve functions in both the *display and interaction* category and in the *storage and communication* category. Does this violate the separation of concerns? No, the desired encapsulation is a reasonable one: a design criterion that forbids useful encapsulations should be sharply questioned. However, a poorly designed implementation of this class might seek to violate the separation of concerns or even perform outrageous hacks of the C++ runtime system. A good implementation will maintain the separation of concerns as illustrated below.

Suppose we have defined a communications class `C` that handles the transfer of images to a remote device and a local display class `D` that displays images on the local workstation. The `net_display` class desired can be designed as follows:

```
class net_display
{
    C* comm;      // note: pointers don't allocate the object
    D* local;     //          they point to.
    int which;    // 0=local display, 1=remote display

public:

    // The first constructor creates a local display;
    // the second creates a remote display
    net_display(<various display params>);
    net_display(char* machine_name, <various display params>);

    // The display function sends the image to comm or local
    // depending on the value of which.
    net_display& display(image im);
}
```

This implementation provides the desired functionality without any hacks, while maintaining the separation of concerns.

4.7 Conclusion: Toward a Collaborative Class Design Methodology

Well-designed class libraries could make possible the software economy [3] that object-oriented design has largely failed to create. The lack of guidance for library design provided by the object-oriented design methodology has allowed library developers to aim either too low (and create collections of tiny classes that are not really useful) or too high (and create comprehensive monolithic hierarchies that attempt to provide too much generality). An intermediate approach is more likely to be effective, especially in the compilation environment of C++.

Effective class libraries will require more explicit design criteria and a preference for larger classes that encapsulate concepts that are really of interest to their users. This paper has criticized three common design criteria and has presented a new candidate design criterion motivated by a particular separation of concerns.

Effective libraries are most likely to be produced in a collaborative effort between the library architect and an application area specialist. The time has come and the tools are at hand to make this kind of collaborative software design feasible. The separation of concerns between architecture, implementation, and realization provided by object-oriented design, and by the structure of C++ classes in particular, elevate the level of discourse about software design to the application level where our clients can effectively contribute. Only by involving application specialists, who are the ultimate users of the library, can essential concepts be identified and effective encapsulations created. Such collaborative efforts keep the library architect focused on issues that yield real progress and ensure that the effort invested in library design will indeed be valuable to a real user.

References

1. Brooks, F. P, "No Silver Bullet: Essence and Accidents in Software Engineering," *IEEE Computer*, vol 20, no. 4, April 1987.
2. Brooks, F.P, *The Mythical Man-Month: Essays in Software Engineering*, Addison-Wesley, 1970.
3. Cox, B. J., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.
4. Freeman, P. *Software Perspectives: The System is the Message*, Addison-Wesley, 1987.
5. Goldberg, A and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley 1983.
6. Gorlen, K. "An Object-Oriented Class Library for C++ Programmers", *Software: Practice and Experience*, vol. 17, no. 12, December 1987, p. 899.
7. Lippman, S. B., *C++ Primer*, Addison-Wesley, 1989.
8. Nye, A., *Xlib Programming Manual for Version 11 of the X Window System*, O'Reilly and Associates, 1988.
9. Parnas, D. L, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, Dec 1972, pp. 1053-1058.
10. Parnas, D. L, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, March 1976, pp.1-9.

11. Rentsch, Tim, "*Object-Oriented Programming*", SIGPLAN Notices, vol. 17, no. 9, September 1982, pp. 51-57.
12. Stepstone Corporation, *ICpack 201 Tutorial Examples Manual*, Stepstone, 1988.
13. Stroustrup, Bjarne, "*The C++ Programming Language*", Addison-Wesley, 1986.
14. Stroustrup, Bjarne, "*What is Object-Oriented Programming?*", Proc. 1st European Conference on Object-Oriented Programming, Paris, 1987.
15. Sun Microsystems, *SunView Programmer's Guide*, Sun Microsystems, 1986.
16. Weinberg, G. *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.
17. Wirth, N., "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14, no. 4, April 1971, p. 221.

5. Managing C++ Libraries

This chapter describes the scheme we have used to manage the development of COOL (under AT&T Cfront 1.2.1). The scheme involves a hierarchical directory structure, and represents dependencies between classes in the library in a globally accessible file 'coolprelude.h'. We also discuss the structure of the description files (makefiles) used with the UNIX 'make' program, some compiler options we have found to be useful in reducing the size of the library (cool.a file), how to minimize recompilation time after trivial changes to the source code of the library, and how to maintain the library for several different computer architectures sharing the source code under NFS.

5.1 The Problems of Library Management

Libraries of C++ classes help users to exploit predefined encapsulations, so they are essential to achieving the goals of code reuse prominent as a justification for object-oriented design. We have encountered and solved several problems in using and managing the COOL library. This scheme is used in the COOL distribution tape.

An appropriate arrangement of subdirectories for a library can simplify the management of the library by its developers and the understanding of the library by its users. We describe in Section 5.2.1 our scheme for organizing the code of our C++ library into subdirectories. Another issue in C++ library management is whether to store member functions in separate '.c' files or to keep all member functions for a class in a single '.c' file. Section 5.2.2 describes the tradeoffs we have discovered between these organizational strategies.

The management of dependencies and header files in a C++ library is an essential problem that, for many potential C++ developers, seriously limits the usefulness of the language. We have developed a scheme that ensures efficient inclusion of required header files while requiring minimal effort from the application and library developers. This scheme is not difficult to use, but it is rather intricate and takes some effort to establish. Once in place, however, maintenance and use of the library are greatly simplified. Section 5.3 covers this issue of dependency management.

A problem we have encountered with AT&T Cfront 1.2.1 is the production of excessively large executable files. This problem was solved, in part, by the use of CC command options that are documented in the Cfront release notes. We describe in Section 5.4 how we use these options to obtain an order-of-magnitude decrease in the size of the library. This section also discusses a simple scheme we use to avoid recompilation of large sections of the library after trivial changes to source files.

Many recently developed workstations operate under the Network File System (NFS) which makes disk files present on remote machines accessible from any machine on the network. Since machines from different vendors having different architectures can share the COOL source code through NFS, we have developed mechanisms for maintaining multiple versions of the COOL object files for multiple architectures based on the same source and makefiles. Section 5.5 describes this scheme.

5.2 Directory Management

Organizing a library in a hierarchical directory structure simplifies development, maintenance, and use of the library by contributing to two desirable goals: (1) Separation of concerns and (2) Information hiding.

Part of the justification for object-oriented design is the formal separation of the concerns of the user, designer, and implementor of a class. In C++, the interfaces between these communities are formalized in the header (.h) file. The public part of the header file is a contract between the user community and the class designer. The entire header file is a contract between the class designer and the implementor. It is only natural that this separation of concerns should be reflected in the large-scale organization of a library as well as in small-scale coding structures. The encapsulation implemented in the "class" concept is reinforced by physically separating the code of different class implementations. A single directory containing copies of the header files helps to reinforce the separation between user concerns, requiring frequent access to headers of many classes, and implementor concerns, requiring access to code and header files of the particular class being implemented.

The humble admission of the limited capacity of short-term memory leads to the technique called information hiding (encapsulation), e.g. [3]. We seek to minimize the amount of data that a software developer or user must manipulate at one time to correctly use the system. This notion is an essential motivation for object-oriented design and should be reflected in the large-scale structure of libraries as well as in the design of code and languages. A suitable directory hierarchy hides code, headers, and administrative concerns that are irrelevant to the current activity. Minimizing the number of names that must be understood or manipulated at once (the "surface area" of a system [2]) is an important technique for simplifying the use of a large body of software.

5.2.1 Library Directory Organization

We organized the library in a three-level directory hierarchy as shown by the simplified example in Figure 1. The library is located under a directory (called "mainlib" in Figure 1) with subdirectories for sub-libraries consisting of groups of related classes. Several of these groups are stand-alone inheritance hierarchies, while others are collections of topically related classes. Each class has its own subdirectory in the sublibrary directory. Of course, a smaller library might be organized without the intermediate level; the main directory might contain the

class subdirectories. Since users most often refer to header files, we have another subdirectory called "headers" that contains copies of all of the header files throughout the library.

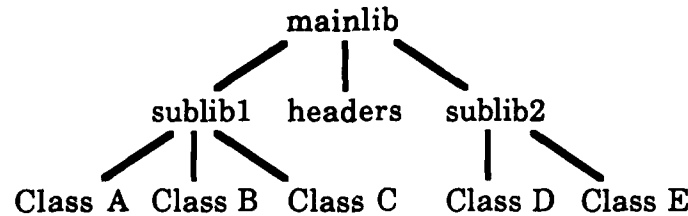


Figure 2: Three-level directory structure used in COOL

Each directory contains a Makefile that performs appropriate operations for a directory at that level. The Makefile in `mainlib` invokes the Makefiles for sublibraries; those Makefiles invoke the Makefiles for class directories. Thus, the directory structure implements a version of the object-oriented philosophy at the level of library organization. All of the Makefiles at each level have a common format (Appendix A) which simplifies addition of new member functions, classes, or sublibraries.

To add a new sublibrary or to add a new class to a sublibrary, the next available symbol (see Appendix A: makefile for Sublib1) should be defined and the comment marks should be removed from subsequent lines involving that symbol. To add a new member function to the class's makefile is even easier: the name of the `.o` file should be added to the OBJ symbol definition. (Note that if the list extends to multiple lines, the escape character `\` must be used at the end of each nonfinal line.)

The third makefile in Appendix A takes advantage of the implicit rules in the UNIX program `make` for transforming `.c` files to `.o` files (see `make(1)`). We modify the implicit rule `.c.o` to echo a status message as each compilation begins.

The Makefile structures described here and in Appendix A are simplified versions of the scheme we are using now. Our present scheme, which will be distributed in a later version of COOL, performs more operations automatically and is even easier to maintain, but it is much more complex, using very advanced features of `make`.

5.2.2 Separating Member Functions

The third Makefile also illustrates another technique we have used to organize and optimize use of the library. We store each non-inline member function in a separate `.c` file. This practice requires that tradeoffs be evaluated to determine whether the technique is worthwhile for a particular case. We believe that for larger libraries the advantages outweigh the disadvantages.

Advantages of storing each function in a separate `.c` file include:

(1) The standard UNIX linker will not attempt to break apart `.o` files. Therefore, if all functions of a class are stored in a single `.c` file, a reference by the application code to any function in the class will cause all of the functions to be linked into the application program's `a.out` file. Separating member functions into separate `.c` files helps to minimize the size of `a.out` files and decrease link time.

(2) A member function can be recompiled without recompiling the entire class. Since this is a common operation, we consider the savings to be significant.

Disadvantages of storing each function in a separate `.c` file include:

(1) Names must be assigned to the `.c` file for each function. This becomes a problem if there exist many overloaded versions of some operations since, in effect, the file name must encode the arguments in order to distinguish the functions. By convention, we name all constructors with an upper-case C followed by an encoding of the arguments to the constructor. `CDest.c` is the destructor, `CNull.c` is the constructor with no arguments, `CCopy.c` is the constructor that takes a reference to an object of the same class, `Cintint.c` takes two integers, etc. If the names assigned to the `.c` files are not unique across the entire library, then when the library is created, you must suppress elimination of duplicate names by using the `-q` option on the `ar(1)` command. Unfortunately, this means that the `.a` file must be re-created from the `.o` files whenever any member function in the library is recompiled. However, by assigning a prefix to the `.o` file names (which can be done in the Makefile), one can sidestep this problem at a small cost in setup overhead.

(2) Compilation of the whole library takes MUCH longer because all relevant header files must be processed for each member function instead of once for each class. We have found this potential disadvantage to be largely irrelevant because we rarely needed to recompile the entire library.

5.3 Dependency Management

The use of libraries in C++ is complicated by dependencies among the classes of the library. An application program must include the header files for all classes on which the application depends, directly or transitively. Direct dependencies are clear from the code of the application program itself. Finding transitive dependencies requires knowledge of the internal structure of the library. We might expect a program author to know and to declare what resources he is using directly, but it is unreasonable to require him to know internal structures of the libraries he is using.

For this discussion, a dependency between classes A and B exists if the header for class A or any member function of class A refers to an object of class B as a member, an argument, or a local variable. (This definition is more conservative but much simpler than the optimal definition for our purpose.)

Due to the complex dependencies between classes in a typical C++ library, many header files may be required even for applications that use objects of just one class. Without management techniques such as we will describe, ensuring that all of the necessary headers are included requires analysis of the entire dependency hierarchy of the library by the library user. We consider knowledge of the internal structure of the library to be an unacceptable burden on the application developer (or on the library developer!). We seek to minimize the interference of such incidental concerns in the development of code that uses the library. Fortunately, we have developed a scheme that ensures that the necessary headers are included while requiring minimal effort from the application or library developer.

An ideal solution to the problems of header file and dependency management would possess the following characteristics:

Whatever is needed gets included.

The users do not pay for what is not needed.

The users do not need to know the entire dependency hierarchy when writing `main()` or member functions.

The system should be easy to use. To make this concrete, we will insist that only one `#include` directive be required in `main()` or member functions.

The solution should support good software engineering practice.

The solution should be compatible with multiple inheritance and other anticipated evolutionary changes in C++.

A program written using our management system should read only the header files that are necessary and should read them only once.

The scheme we have developed conforms to these objectives and allows enough flexibility to handle unforeseen situations with grace.

5.3.1 Dependencies

Consider the small inheritance hierarchy given in Figure 3. Class `foobar` is a base class with derived classes `foo` and `bar`. Class `baz` is not part of the inheritance hierarchy, but since class `foobar` uses objects of class `baz`, any compilation of `foobar` requires inclusion of the header for class `baz`. The dependency structure of these classes is shown in Figure 3.1.1. Notice that only direct dependencies are drawn; transitive dependencies (`foo` requires `foobar` which requires `baz`) are not drawn. (If there were a direct dependency between `bar` and `baz`, for example, we would include that link in the dependency hierarchy graph.)

Inheritance Hierarchy



Dependency Hierarchy



Figure 3: Inheritance and Dependence Hierarchies

The determination of which header files to include when compiling an application or a member function depends on the dependency hierarchy, which depends on internal details of the design of the whole system of classes, most of which is embodied in the header files. The creation of such dependencies is essential to the library's usefulness. If objects are to work together at all, and if code is to be reused at all, then dependencies must exist. We need a method for declaring direct dependencies and reliably tracing those dependencies throughout the dependency hierarchies when needed.

5.3.2 Rejected Approaches

Include what you need.

In this approach, each member function and each application program must contain `#include` directives to obtain whatever is needed. This requires the application developer to understand the entire dependency structure of the library, which we find unacceptable. Furthermore, this approach leads to a long list of `#include` directives, whose creation interferes with the task of software development.

Include everything.

We considered `#include`-ing everything, but this violates the objective of not paying for what is not needed. In a large library, the time required to process all of the `.h` files is not negligible, so we reject this option.

Use `#ifndef` SYM ... `#endif` chains. (Sometimes called "wrappers")

A common solution in practice requires surrounding each `.h` file with C preprocessor directives to test whether a symbol unique to that class is defined and if not to process the header file. If the symbol is defined, the translator must still scan the file until reaching the `#endif` at the end of the file. This is a plausible solution, which we finally rejected for several reasons. First, this system allows a header file to be `#included` many times - it will be processed only once, but we prefer that it not be touched at all if it is not required. Second, this approach requires that the user know the path names to many header files - details that are incidental to the coding task and should be eliminated from the programmer's concern. With our hierarchical directory scheme described in

Section 2.1 specifying path names requires that users know the whole directory hierarchy, which we find unacceptable. Third, we find the intrusion of the `#ifndef...#endif` directives in our header files aesthetically displeasing. We prefer a less invasive approach that isolates administrative concerns, such as dependencies, in a single location away from our code.

5.3.3 Proposed Solution

The solution we have developed is noninvasive, it requires no knowledge of the dependency structure or the directory structure of the library, and it causes header files to be included only when required, and only once even then. The following sections explain our scheme.

5.3.4 Dependency Files

In the subdirectory for each class, we define a dependency file (with a `.d` suffix) that declares direct dependencies on other classes by defining symbols of the form `D_classname`. After the appropriate symbols are defined, we check to see whether the "prelude" for the whole library has been defined; if not, we `#include` it. (The structure of the prelude is critical to our scheme and will be discussed in the next section.)

For the example given in Figure 3.1.1, the dependency file for classes `bar` and `foobar` are shown below.

| FILE <code>bar.d</code> | FILE <code>foobar.d</code> |
|---|---|
| <code>#define D_BAR</code> | <code>#define D_FOOBAR</code> |
| <code>#define D_FOOBAR</code> | <code>#define D_BAZ</code> |
| <code>#ifndef D_PRELUDE</code> | <code>#ifndef D_PRELUDE</code> |
| <code>#include "../libprelude.h"</code> | <code>#include "../libprelude.h"</code> |
| <code>#endif</code> | <code>#endif</code> |

The structure of the dependency file is determined entirely by the dependency structure of the library. Typically, a class will declare a dependency on itself, its base class if any, and the classes referenced by the class as arguments to messages or as local variables in member functions.

Other additions to the `.d` file can handle special situations. If there are classes with mutual dependencies, the forward declaration of the sibling classes can be inserted in the `.d` files of each class. The `#includes` for header files of specialized libraries may be inserted in the dependency file. In our library, for example, header files for the SunView libraries are `#included` in the `.d` file of the `imagetool` class which handles image display on Sun Workstations.

The ability to place `#include` directives for special `.h` files in the `.d` file (thereby placing the `#include` in every compilation involving the `.h` file of the class) does not preclude the option of placing `#include` directives for some system libraries

in the specific member functions that require them, or even placing additional `#include` directives in the `.h` file itself.

5.3.5 The Prelude File

In the main directory for the library, we define a 'prelude' file which has three parts. The prelude file for the example in Figure 3.2.1 is given below. The first part of the prelude file `#includes` system header files that we want always to be included. (But in the future those libraries might be organized similarly to ours.) The second part is a level-by-level traversal of the dependency graph from the top down in which the `.d` files of all classes that have been declared as being required are `#included`.

The top-down traversal is critical to allow all of the transitive dependencies to be recognized correctly. For example, if the application program references only class `foo`, we know to include `foo.d`, which contains the definition of `D_FOOBAR`. Since we are going top-down through the dependency hierarchy, we will *later* check `D_FOOBAR` and `#include` `foobar.d` which defines `D_BAZ` and so on.

In the third part of the prelude file, the header files of all classes that have been declared as needed by defining the `D_classname` symbol are `#included`, once and once only. The classes are checked in bottom-up order according to the dependency hierarchy so that every `.h` file that is required is defined before it is needed by another class definition. Thus, all of the `.h` files that are needed are included, and they are included only once. A prelude file for the example given in Figure 3 is shown below.

FILE `libprelude.h`

| | |
|--|---------------------------------|
| <code>#define D_PRELUDE</code> | Define the prelude symbol. |
| <code>#include <stream.h></code> | |
| <code>#include <string.h></code> | Include system header files |
| <code>#include <math.h></code> | that are always needed. |
| | |
| <code>#ifdef D_FOO</code> | |
| <code>#include "../foo/foo.d"</code> | Include '.d' files in top-down |
| <code>#endif</code> | order traversal of dependency |
| <code>#ifdef D_BAR</code> | hierarchy. This determines |
| <code>#include "../bar/bar.d"</code> | what headers are required |
| <code>#ifdef D_FOOBAR</code> | using just compiler symbols. |
| <code>#include "../foobar/foobar.d"</code> | |
| <code>#endif</code> | |
| <code>#ifdef D_BAZ</code> | |
| <code>#include "../baz/baz.d"</code> | |
| <code>#endif</code> | |
| | |
| <code>#ifdef D_BAZ</code> | |
| <code>#include "../baz.h"</code> | Include '.h' files in bottom-up |
| <code>#endif</code> | order traversal of dependency |
| <code>#ifdef D_FOOBAR</code> | hierarchy. |
| <code>#include "../foobar.h"</code> | |

```

#endif
#ifdef D_BAR
#include ".../bar.h"
#endif
#ifdef D_FOO
#include ".../foo.h"
#endif

```

The prelude file looks more complex than it is. Maintaining the prelude file is also easier than it looks. Classes at the same level in the dependency hierarchy can be listed in any order, so the ordering of the sections is not as critical as it might appear. Also, the most common situation requiring modification of the prelude file is the implementation of a new class, which happens relatively infrequently compared to changes in member functions. We have found that development of new classes outside the library is a safe and effective strategy. The classes can be incorporated into the directory structure and the prelude file as they mature.

It is possible to develop tools for automatically producing the prelude file for a library based on the .d files or on a separate description of dependencies. We have not pursued this possibility because we have never found a need to regenerate a prelude file.

5.3.6 Using the System

To use our strategy for managing dependencies, the application programmer need only `#define` the symbols for the classes used in the program and `#include` the library prelude file. The writer of member functions must simply include the class's dependency file. Examples are given below:

| Application Code | Member Function |
|--|---|
| FILE prog.c | FILE foo::reset.c |
| <pre> #define D_FOO #define D_BAZ #include ".../mainlib/prelude.h" <code of prog.c> </pre> | <pre> #include "foo.d" <code of reset.c> </pre> |

We have found this to require a minimal level of invasion in the process of preparing a .c file, and while our scheme is rather costly to set up, it is very easy to maintain. The use of more than one library configured as we describe also involves minimal intrusion in the user's code (one `#define` statement per class used and one `#include` statement for the prelude file of each library).

5.3.7 Robustness

Our organization scheme degrades gracefully in the presence of several kinds of likely errors.

If the programmer omits a `D_classname` symbol definition for a class used directly in the application code and it so happens that the class whose symbol was omitted is required by another class that *is* listed, then everything works normally and there is no error. Thus, if the programmer does know something about the structure of the library, he can take advantage of that knowledge and minimize the administrvia in the `.c` file.

If the programmer omits a symbol definition for a class that is indeed required, the C++ translator will flag syntax errors claiming that "foo is not a class name" when you know very well that it is.

If the programmer omits the `+e1` flag on the compilation and links with a library compiled with `+e0`, the linker will give error messages like "`__foobar_vtbl__` is not defined".

If the checks for a new class are placed in the first (second) traversal in the prelude file anywhere above (below) the highest (lowest) existing class used by the new class, then everything will work correctly, at least until the next class is entered.

5.4 Optimizations

The scheme presented so far virtually eliminates the need for application programmers and library developers to separately maintain knowledge about the internal structure of the library. And although this scheme reduces incidental tasks significantly two troublesome problems, common to all larger libraries written in C++, remained. First, the size of the library's object file was very large compared to the cumulative size of the source for the library. The excessive volume propagated to the application code resulting in executable modules of application programs that were also unacceptably large, and this occurred even when the application code used only a few routines from the library. Section 5.4.1 describes how we optimize the size of both the library's object file and the size of application program executables. Second, the dependencies within the library cause recompilation time to be high (many source modules were included in the recompilation) even when small, trivial changes were made to class specifications (`.h` files). Section 5.4.2 proposes a simple scheme to reduce the number of recompiled files when trivial changes are made.

5.4.1 Size Optimization

Libraries are essential productivity aids that multiply the advantages of object-oriented design by making possible the reuse of separately compiled code. When we began using AT&T cfront 1.2.1 with our library, we found that the library generated a huge object file (about 2Mbytes for 10K lines of code) and most programs linked with the library produced unacceptably large executab^l files of over 1Mbyte each. Another library for similar purposes but written in C produces executable files of 150K or less. The exorbitant size of the C++ executables made the language unacceptable for routine use.

The Cfront 1.2.1 release notes describe the operation of a new compiler option, `+e`. Basically, `+e` controls whether virtual function tables are created separately for each `.c` file or are shared among all of the `.c` files. This option is crucial for library developers and users since it produces massive reduction in the size of executable files, and as side benefits, significant decreases the time required for library creation and linking.

We reproduce in Appendix B the description of the `+e` option from the C++ Translator Release 1.2 Addendum to the Release Notes [1]. What is lacking in them is the simple description of how to use `+e` for managing C++ libraries. The critical insight is as follows:

Compile all of the routines in your library with `+e0`, and compile programs using the library with `+e1`.

In COOL, which consists of about 60 classes, the use of the `+e` option caused the library's object file to decrease from about 2Mbytes to about 200Kbytes - an order of magnitude reduction! Programs linked with the library decreased in some cases from 1.5Mbytes to 500Kbytes or less, with an average of 30% reduction even for simple programs using only a few primitive classes. Significant reductions have been observed in the time required to create the library from `.o` files, to execute `ranlib(1)` on the library, and to link programs using the library. The use of the `+e` option transformed COOL from an interesting curiosity to a usable tool.

5.4.2 Minimizing Recompilations

Using the comparison of timestamps and the dependency structure to determine the set of files that need recompilation after an arbitrary change to a C++ source file hinders the incremental development of large systems in C++ since a single change can unleash a torrent of recompilations. We would like to see a method that could (1) determine the minimal set of files that need recompilation given some arbitrary change and (2) produce a list of files that need changes as a consequence of the original change. We are not aware of any method presently capable of either of the above. In the absence of a method for determining the minimum set of files that need recompilation, we propose here a reasonable alternative.

Our method depends on the library developer distinguishing two kinds of source code modifications: trivial and non-trivial. Trivial changes do not necessitate any recompilation, and non-trivial changes do necessitate recompilation of the files which are dependent on the file to which the change has been made.

The problem of determining the minimal set of files that need recompilation cannot be satisfactorily solved by looking at the dependency structure of the library. For example, suppose that `C.c` includes `A.h` and `B.h`, and assume that `A.h` has been changed in a trivial way (that would not require `C.o` to be recreated) and `B.h` has been changed in a non-trivial way (that would require `C.o` to be recreated). If some method based only on dependency information and

timestamps for determining which files required compilation is invoked at this point, it will update the timestamp of C.o without recompiling since C.o depends on A.h and the change to A.h is known to be trivial. But this is an error since a non-trivial change was made to B.h. A request to remake the system at this point would not remake C.o since its timestamp indicates it is newer than B.h. Given a sequence of interleaved trivial and non-trivial changes to the system, dependency information is inadequate to determine a minimum recompilation set. If just the trivial change had been made, then basing recompilation on dependency information would work, but since only trivial changes have been made, even this is not necessary.

The above example leads to the observation that if a sequence of trivial changes has been made to a library, the timestamps of all of the .o files in the system can be updated (e.g. using touch(1)) to the current time without loss of consistency. Consider the state of the system before the sequence of trivial changes; the system is in a 'no recompilation needed' state. By definition, the trivial changes have not really changed this state so it can do no harm to touch all of the .o files in the system since this action will return the actual state of the system to 'no recompilation needed'.

With this in mind we have included a "trivial" target in the Makefiles of the library (see Appendix A). This could be accomplished, of course, by a very small shell script, but we prefer to incorporate the operation in the Makefile system because of the added control afforded. We can invoke the updating at any of three subdirectory levels. Invoking at the mainlib level will update the complete system, at the sublibrary level will update related classes and at the class level will update only a single class.

5.5 Conclusion

We have described several methods that when used together reduce the complexity of using and maintaining a large library of C++ routines. Our scheme includes structuring the directories containing the source code of the library in a three level hierarchy, using a separate source file for each non-inline member function, explicitly encoding the dependency structure of the library in '.d' files and the prelude file, and optimizing the size and recompilation of the library. We feel that this scheme has changed our library from an interesting academic exercise to a usable tool. Also, the scheme maintains conceptual integrity in the presence of multiple libraries.

References

1. C++ Translator Release 1.2, *Addendum to the Release Notes*, AT&T, 1986, select code 307-175, pp. 11-12.
2. Cox, B.J., *Object Oriented Programming An Evolutionary Approach*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986, pp. 16-17.

3. Parnas, D., On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM, Volume 15, Number 12, December 1972.

Appendix A

MAKEFILE FOR MAINLIB

```
.SILENT:

A = sublib1
B = sublib2
#C =
#D =

trivial:
    echo "Update the complete system"
    (cd $A; make trivial)
    (cd $B; make trivial)
#    (cd $C; make trivial)
#    (cd $D; make trivial)
    echo "Update complete"
compile:
    echo "Perform all compilations"
    (cd $A; make all)
    (cd $B; make all)
#    (cd $C; make all)
#    (cd $D; make all)
    echo "Compilations complete"

cleanup:
    echo "Cleanup all libraries"
    (cd $A; make cleanup)
    (cd $B; make cleanup)
#    (cd $C; make cleanup)
#    (cd $D; make cleanup)
    echo "Cleanup complete"

create:
    echo "Create mainlib.a from scratch"
    (cd $A; make library)
    (cd $B; make library)
#    (cd $C; make library)
#    (cd $D; make library)
    touch mainlib.a
    rm mainlib.a
    mv newlib.a mainlib.a
    ranlib mainlib.a
    echo "mainlib.a complete"

MAKEFILE FOR SUBLIB1
.SILENT:
.in +1
LIB = SUBLIB1
```

```
MAINLIB= /.../mainlib/newlib.a    ( <-- full path name of the .a
file)
```

```
A = ClassA
```

```
B = ClassB
```

```
#C =
```

```
#D =
```

```
trivial:
```

```
    echo "$(LIB) Update the sublibrary"
```

```
    (cd $A; make trivial)
```

```
    (cd $B; make trivial)
```

```
#    (cd $C; make trivial)
```

```
#    (cd $D; make trivial)
```

```
    echo "$(LIB) Update complete"
```

```
all:
```

```
    echo "$(LIB) Begin Compilation"
```

```
    (cd $A; make all)
```

```
    (cd $B; make all)
```

```
#    (cd $C; make all)
```

```
#    (cd $D; make all)
```

```
    echo "$(LIB) Compilation Complete"
```

```
cleanup:
```

```
    echo "$(LIB) Begin Cleanup"
```

```
    (cd $A; make cleanup)
```

```
    (cd $B; make cleanup)
```

```
#    (cd $C; make cleanup)
```

```
#    (cd $D; make cleanup)
```

```
    echo "$(LIB) Cleanup Complete"
```

```
library:
```

```
    echo "$(LIB) Create library"
```

```
    ar lq $(MAINLIB) $A/*.o
```

```
    ar lq $(MAINLIB) $B/*.o
```

```
#    ar lq $(MAINLIB) $C/*.o
```

```
#    ar lq $(MAINLIB) $D/*.o
```

```
    echo "$(LIB) Create library complete"
```

```
MAKEFILE FOR CLASS CLASSA
```

```
.SILENT:
```

```
.in +1
```

```
CLASS = CLASSA
```

```
OBJ = CDest.o CNull.o Cintint.o reset.o draw.o compute.o
```

```
trivial:
```

```
    echo "$(CLASS) Update"
```

```
    touch *.o
```

```
    echo "$(CLASS) Update complete"
```

```
all:
```

```
    echo "$(CLASS) Begin Compilation"
```

```
    make $(OBJ)
```

```
    echo "$(CLASS) Compilation complete"
```

```
cleanup:
    echo "$(CLASS) Cleanup"
    /bin/rm *.o
    echo "$(CLASS) Cleanup complete"
```

```
CC=CC
CFLAGS= +e0 -fswitch
.c.o:
    echo "Begin $*.c"
    $(CC) $(CFLAGS) -c $<
```

Appendix B

Reproduced from the AT&T C++ Translator Release 1.2 Addendum to the Release Notes [1]

CC +e Option

There is a new compiler option, +e. It works as follows: CC +e1 causes virtual tables to be external and defined, that is, initialized. CC +e0 causes virtual tables to be external but only declared, that is, uninitialized.

CC causes virtual tables to be local to a file (static) and defined.

For example, given a header file, SUPER.h, with the class definition

```
class SUPER {
public:
    virtual size();
    virtual is_equal();
    virtual grow();
};
```

each separately compiled executable component with a #include of SUPER.h will normally generate a static virtual jump table. Using +e allows you to optimize a program by ensuring that only one virtual table is generated per class. This can save 25 percent of object files size, a.out size, link time, and (in the case of cross-compilation) download time.

For example, suppose our executable is composed of t0.c, t1.c, t2.c and t3.c, each containing a #include of SUPER.h. The target portion of our makefile would look as follows:

```
foo: $(OBJECTS)
    $(CC) -o foo $(CCFLAGS) $(OBJECTS)

t0.o:    SUPER.h
    $(CC) $(CCFLAGS) -c t0.c

t1.o:    SUPER.h
    $(CC) $(CCFLAGS) -c t1.c

t2.o:    SUPER.h
    $(CC) $(CCFLAGS) -c t2.c

t3.o:    SUPER.h
    $(CC) $(CCFLAGS) -c t3.c
```

To generate only one instance of the SUPER class virtual table, the makefile would be changed as follows:

```
foo: $(OBJECTS)
      $(CC) -o foo $(CCFLAGS) $(OBJECTS)

t0.o:      SUPER.h
      $(CC) $(CCFLAGS) -c +e1 t0.c

t1.o:      SUPER.h
      $(CC) $(CCFLAGS) -c +e0 t1.c
t2.o:      SUPER.h
      $(CC) $(CCFLAGS) -c +e0 t2.c

t3.o:      SUPER.h
      $(CC) $(CCFLAGS) -c +e0 t3.c
```

The +e1 option in the t0.o file suppresses the static scope specifier in the generated t0.o object.

The +e0 option in the other targets suppresses the generation of the SUPER class virtual tables.

6. Implementation Techniques in COOL

Research efforts in image pattern recognition and computer graphics face two kinds of software problems. First is the intrinsic complexity of the algorithms developed in the course of the research. Second is the design and construction of the researcher's software toolbox including fundamental operations, standards for data storage and communication, and interfaces to rapidly changing sets of display and interaction devices. The complexity problem is intrinsic to the subject matter and objectives, and it is the proper domain of the researcher. The second kind of problem is incidental to the research objectives and can be addressed by adoption of modern software development tools and disciplines, including object-oriented design for code and hypertext structures for documentation.

This section describes several techniques we have developed while designing COOL, an integrated object-oriented software workbench for image pattern recognition and interactive computer graphics. Design criteria for the system include (1) pervasive integration of constructs, (2) maximum flexibility for researchers using the system, (3) minimum user effort to invoke the facilities, and (4) purity of the object-oriented design. We will describe in this chapter object-oriented techniques we have developed for managing massive data structures, providing type-independence at the user level, encapsulating device dependencies, processes, and class interfaces, and designing the required software system while maintaining integration of concepts.

6.1 Managing Massive Data Structures

The kinds of structures we manipulate (images, pattern matrices, graphical models) often have large or very large memory requirements. We do not want to reallocate, copy, and destroy these large structures in each function call and function value return. Instead, we implement large structures using an intelligent pointer object that contains a pointer to a storage object that contains the data. The function of the intelligent pointer object is to pass most messages it receives on to the storage object where code for performing the requested action resides.

The class `image`, for example, contains only a pointer to an object of class `buffer` which contains the pixel data for the image as well as the shape of the buffer and the shape and position of a rectangular window for use as an area of interest. Class `image` understands many messages, but the action of `image` is simply to forward the message to the `buffer` object for processing. For example, the member function

```
image& image::add(image&)
```

is defined in the `image` class, but its action is simply to invoke the `buffer` method

```
void buffer::add(buffer*).
```

Objects of class `image` are physically small, so they may be copied, allocated, and destroyed as needed with negligible performance penalty. Therefore, we need not

hesitate to pass an image as a value parameter to a procedure if that is convenient.

We must be careful, however, in the constructors and destructor of image to avoid repeated allocation, copying and deallocation of large buffer objects. Thus, the `image::image(image&)` and `image::operator=(image&)` messages simply copy the buffer pointer of the source image and do not allocate a new buffer. In order to prevent the destructor `~image()` from deallocating the buffer while it is still being used by another image object, the image destructor checks a reference count in the buffer object and deletes the buffer only if the reference count is zero after decrementing. This reference count mechanism also prevents the large buffer objects from being left in the heap as garbage, which would soon result in exhaustion of virtual memory. We are planning to use the intelligent pointer class/storage class implementation for all of our large structures such as pattern matrices and some graphical models.

6.2 Implementing Type Independence

The separation of intelligent pointer and storage classes also makes possible type independence at the user level. Storage types for images supported in COOL are BYTE, INTEGER (short), REAL (float), and COMPLEX (a pair of floats). Objects of class image are manipulated in natural ways, without any special consideration of their storage type. Messages involving the pixel data are passed to the image's buffer object. The buffer class contains virtual function declarations for the suite of operations affecting the pixel data. The real work is finally performed in subclasses of buffer that are type-specific: `byte_buffer`, `int_buffer`, `real_buffer`, and `complex_buffer`. The virtual function mechanism allows the image to send messages to a buffer object without regard to the storage type of the buffer.

Necessary storage type conversions can be explicitly requested using the message `image image::convert(buffer_type, convert_type)` but appropriate conversions will take place without programmer intervention. When more than one kind of conversion between a pair of types is possible (complex-to-real can be performed by real part, imaginary part, magnitude, or phase), a default (magnitude conversion) is assigned and an optional parameter can be used to override the default.

The consequence for the programmer of this fairly complex internal decoupling is that image objects can be manipulated naturally in expressions such as

```
image result=im1.add(im2).add(im3);
```

without the error-prone distractions of creating and dereferencing pointers and without the concerns over matching the storage types of the various images. These internal facilities (virtual functions, reference counting, intelligent pointer classes) are completely hidden from the user's attention.

The overhead of the extra level of indirectness required to implement these features is negligible. Processing massive objects takes considerable time anyway, so since we get both type-independent manipulations and a reasonable

method for managing the large memory requirements, we do not begrudge a little overhead cost.

6.3 Encapsulating Device Dependencies

A serious problem in a high-technology lab such as ours is keeping the software base current and consistent with the available hardware capabilities. We have experienced a phenomenon we call "hardware indigestion" in which we have difficulty incorporating new hardware into existing projects because of software incompatibilities between the various devices. New equipment must be carefully assessed to determine whether the effort required to make it compatible with and usable in our environment is worth the time and effort involved. Promising machines have been left to sit idle because of the difficulty of integrating the machine with the rest of the laboratory. Advances in graphics and imaging devices have usually involved incremental speed and resolution enhancements, not entirely new functionality. Accessing that functionality is often difficult because the vendor's software involves intricate code that is optimized in some ways for the device's capabilities and that is not amenable to incorporation into a uniform, laboratory-wide interface.

We have developed three kinds of device encapsulations. Disk file manipulations are supported by a class `diskfile` that handles basic disk operations such as open, close, read, write, and seek. These operations are then invoked by derived classes that "know" the structures of particular kinds of disk files such as `imagefile`, `polyfile`, and `textfile`. New kinds of file structures can be added as derived classes of `diskfile` without changing any of the existing code.

Another kind of device encapsulation we have developed involves analog input devices such as knobs, joysticks, and sliders. The key to the design of these classes was recognizing that the only differences between them from the system's viewpoint are the name of the device driver and the number of bytes expected from the device in a single read operation. Operations provided by the abstract superclass `analog_device` include `poll()` to force a read of the device and `int rawdata(int)` to obtain one of the values provided by the A/D converter. A uniform user interface is provided by adopting the convention that the device-specific classes convert the integer raw data value to a double between 0.0 and 1.0. Now the roles of the devices can be interchanged by simply declaring the device object to be of a different device subclass. Device-specific interfaces are also available; a 2-D joystick can return a point, and a 3-D velocity joystick can return a vector. Work is underway in the laboratory to produce more generic hardware interaction devices that would be more interchangeable but would still present a uniform interface to controlling software.

The third kind of device encapsulation involves display devices. In COOL, the unit of encapsulation for display devices is the viewing surface, so on window-oriented systems, the display object created is a window. Thus, several display objects may be active at once on a device. Several basic capabilities are defined including clearing the display, drawing lines, writing text, rendering polygons, and

displaying images. We have not satisfactorily solved the problem of incorporating enhanced capabilities of some devices, especially when the architecture of the device requires the data to be prepared differently for display processing as is the case for vector versus raster displays or specialized graphics architectures such as the PixelPlanes graphics engine developed in our department.

Another kind of device encapsulation that has been explored but is not yet fully integrated into COOL is encapsulation of other processors on the network. This encapsulation was originally developed to support compute-intensive processes and will be discussed in the next section.

6.4 Process Encapsulations

Since our research involves development of new algorithms for imaging and graphics problems, encapsulation of these frequently-changing processes is essential to our research software environment. We have used a technique called process encapsulation to simplify the use and invocation of the processes based on the dictum *Encapsulate most deeply that which is most likely to change*.

Conceptually, a process encapsulation creates an object that we call an enzyme or a catalytic object, whose purpose is to mediate interactions among other objects. In a process encapsulation, a class structure is defined for the process type, a renderer or a classifier, for example, that specifies the minimum functionality and parameters of such a process. Then, specific algorithms are defined as derived classes with their own parameters, as required. To use the process, we create an object of the desired subclass, connect it to other objects and supply the parameters it needs, and send it a "begin" message. The objects that supply input and output services to the process encapsulation object can be supplied in three ways: by arguments to the constructor, by assignment in separate messages to the object, and by arguments to the "begin" message. This design allows the user to customize the process in a separate code segment from that where the process is invoked, leading to clean, readable code in the routine that invokes the process. The inheritance of fundamental operations and structures from the process' base class contributes to rapid development and evaluation of algorithm modifications and parameters.

The best example of a process encapsulation in COOL is the `fft_server` class. This class performs fast discrete Fourier transforms and inverse transforms on images. To use the `fft_server` class, one creates an `fft_server` object that is specific to a particular image size. If FFTs of images of a different size are needed, a separate `fft_server` object is required. The constructor of `fft_server` computes a series of tables that are used to make the actual FFT computations blazing fast. The size of these tables depends on the image size, so different image sizes require different `fft_server` objects. The `fft_server` is an excellent example of a process that is well-defined and of interest itself and therefore a candidate for a process encapsulation.

Another interesting encapsulation is our `Convex` class, under development for a later release of COOL, that encapsulates interfaces and operations on a Convex mini-supercomputer accessible from our laboratory over a network. Since software must be running at both ends of the network connection, and since the Convex does not support C++, we have not managed to make this connection simple to use. However, we have managed to make use of the Convex for massive filtering operations related to our computer vision research. The `Convex` class must partition the image data into chunks to send over the network, inform the server procedure on the Convex that the transfer is complete, and wait for word that a result is ready. The local workstation and the Convex computer's software handshake to transfer several massive result images back for further analysis. Writing the `Convex` server involved more tedious byte-ordering manipulations than we thought warranted massive continued effort once the initial project was completed. We will attack it again later and try to resolve the ease-of-use problems. We would like to develop code to interface easily to our Pixar Image Computer as well, but there has been insufficient interest in using that machine for anything more sophisticated than as a large frame buffer, for which the Pixar's software is adequate.

6.5 Class Interface Encapsulations

The definition of process interfaces is facilitated by classes that store intermediate results in a standardized form (or a set of agreed-upon forms). A typical example occurs in a graphics pipeline where various kinds of object models must be converted into "rendering primitives" that the display devices understand and can process. By adopting a standard set of rendering primitives, developers of process encapsulations for renderers and developers of display device encapsulations are insulated from each other's internal data structures and processing requirements. Development of renderers can proceed in a device-independent fashion and augmentations to the set of rendering primitives are explicitly noted and handled by all device encapsulations.

The interface between stages in the graphics pipeline is provided in COOL by the `polyobject` class. A `polyobject` is a list of `Polylines`, which are matrices containing points. `Polyobjects` are used to store object prototype models, intermediate results of viewing transforms, clipping results, and `polylines` ready for display. `Polyline` objects can hold polygons, but the additional semantics must be enforced by programming convention; there is no distinction supported in COOL.

By standardizing interface classes, most of our graphics research efforts can begin to share code. We find that research that is advancing the state of the art sometimes still must diverge from the standards either in order to optimize performance or in order to explore new paradigms that are beyond the state of the art. An example of the former case is real-time interactive graphics using customized parallel architectures requiring a different structure in the graphics pipeline. An example of the latter case is research in texture mapping in which the graphics pipeline is modified to accommodate an entirely new kind of rendering.

In order to accomodate these research efforts, the COOL graphics classes are designed for ease of use by system developers and has been criticized for being less than optimal for users. This is an explicit design tradeoff that we have accepted in order to provide flexibility and control at the expense of some ease of use and fidelity to user-level conceptual structures.

6.6 Separation of Concerns

The interface classes described above are used to implement a separation of concerns that has guided the design of our basic class structures. We will illustrate its effect with an example from our image processing library.

In our first implementation of class `image`, we included messages such as `load`, `save`, and `display`. The resulting structure had several problems. First, putting everything into `image` made the code too large. Second, the code for `image` had an unpleasing asymmetry. The code for the `load`, `save`, and `display` messages overwhelmed the code for the numerous image processing messages, most of which were less than ten lines each. Third, the intricate code we worked out for handling disk I/O for images was unusable by any other classes, and the `display` operation was useless on any but the device we defined it for.

The next incarnation of `image` separated the concerns of storage, processing, and interaction devices into different classes. Storage was handled by a class, `diskfile`, that encapsulates all low-level disk operations but without any knowledge of the semantics of the file being manipulated. A subclass `imagefile` directs the decoding and interpretation of the disk data. The `image` class retains the processing operations. Display of images was moved out to an `imagetool` class. Thus, an "image" became a "rendering primitive" that all display devices are expected to process in some reasonable manner.

The principle of separation of concerns is primarily an implementation principle that helps to provide the control and flexibility that we need in our research environment, but it sometimes works against the kind of user-level ease-of-use and fidelity to conceptual structures that is a hallmark of Smalltalk. We are still investigating whether and how a user-level class structure might be imposed atop our implementation structures without redesign for each alternative implementation of a graphics or imaging pipeline.

6.7 Conclusion: Is C++ Really the Right Tool?

Object-Oriented Programming is a code packaging discipline that imposes a reasonable structure on large bodies of code, with additional benefits of code sharing within each class hierarchies and effective conceptual metaphors for talking and thinking about programs. Object-oriented programming provides just the kind of discipline and structure that we need as the size and complexity of our software base increases beyond a level where a single person can maintain, control, and understand it.

Since our laboratory is an established C and UNIX environment, a language derived from C makes sense, especially in view of our large installed base of C code and our need for implementation control in order to support real-time operations and efficient handling of large storage structures. These properties of our environment and objectives make C++ a reasonable language for our software development efforts.

7. Program Examples

This chapter will describe how to write programs using COOL and illustrate some useful programming techniques with examples.

7.1 A Simple Program Using COOL

```
#define D_IMAGEFILE
#include "coolprelude.h"

image build_image(int size)
//
// This procedure creates an image of a specified size.
{
    debug("Begin image create loop");
    image im(size,size,REAL);    // allocate the image
    subscript s(0,0,im.shape()); // set up for loop
    for(s.init(); s.test(); s++)
    {
        im.set(s,s.size()%256);    // define each pixel
    }
    debug("Image create loop done");
    return im;
}

main() // Program MakeImage
//
// This program gets an image file name and row size,
// creates an image, and saves it to the named imagefile.
{
    int size;
    char fname[80];
    cout << "Image file name:"; cin >> fname;
    cout << "Image row size="; cin >> size;

    imagefile* imfile;
    subscript s(size,size);
    imfile = new imagefile(fname,REAL,s,1); // create the disk file

    debug("Begin build_image");
    image im=build_image(size); // build the image in memory

    debug("Save the image");
    imfile->save(im);            // save the image to the disk file

    debug("Close and exit");
    delete imfile;
}
```

The simple but complete program above illustrates several important techniques involved in using COOL successfully. The program's purpose is to create a REAL image of a user-specified size using a procedure called `build_image`. This particular `build_image` procedure will produce a series of diagonal intensity ramps. The following discussion will explain each section of the program.

The first lines of the program define symbols for the C++ compiler indicating what COOL classes are required by this program. The form of the symbols is `D_<class_name>` where the class name is in all upper case. Once the appropriate symbols are defined for classes used directly by the program, we

```
#include "coolprelude.h"
```

which ensures that all other `#include` files from COOL that are needed to fully define this program will be included at the appropriate points in the compilation. Thus, all COOL classes that are required by the `imagefile` class will be included by the `coolprelude.h` file. Note that you must specify the correct path for your version of `coolprelude.h`; this depends on your where your local version of COOL is installed. At UNC, the correct path name is

```
"/usr/local/include/cool/coolprelude.h".
```

Of course, it is possible to use the `-I` compiler switch to specify the directory. Or, at UNC, the prelude file is in the compiler's default search path, so we could write

```
#include <cool/coolprelude.h>
```

Alternatively, you may wish to define a soft link to `coolprelude.h` in some directory of yours that you know is in your search path.

The `build_image` procedure illustrates that C++ procedures may accept arguments and return objects as required. The debug macro in COOL simply prints its string argument. The declaration of the image `im` specifies that the size will be 256x256 pixels and that the storage type of the associated buffer is REAL (float). The subsequent declaration of the subscript object `s` invokes a special constructor that prepares the subscript for use as the control for the following `for` loop. Messages to the subscript object cause it to initialize, test, and increment itself, hiding from the programmer's view the details of the row and column structure of the image object. The value assigned to the image at location `s` is the product of the row and column subscripts (`s.size()`) modulo 256. The image is returned as the value of `build_image`.

The `main()` procedure prompts the user for a file name and an integer size for the image to be produced. It then creates the `imagefile` that will hold the image by specifying the file name, buffer type, size, and number of planes. Notice that the `imagefile` object was allocated on the heap by using the `new` operator. This allocation method gives more control over the time and method of deallocation than does automatic allocation. For `imagefile` objects, this means that we can control when the file is closed (since the `imagefile` destructor simply closes the file). In this particular instance it makes no difference in the action of the program. A local image object is allocated and assigned the value returned by `build_image`. This result is saved to the `imagefile`. Before exiting, this program explicitly closes the `imagefile` by using the `delete` operator on the `imagefile` pointer. This does not delete the file; it deletes the `imagefile` object. The destructor of that object simply closes the file.

7.2 Preparing a Program Using COOL

To compile and link a program `Myprog.c` that uses COOL, you may use a Makefile such as the one included below (based on the GNU `gmake` program):

```
.SILENT:
CC=CC
COOLBIN= /usr/local/lib
COOLLIB= $(COOLBIN)/cool.a
COOLINC= /usr/local/include/cool
CFLAGS= -c +e1 -I$(COOLINC)
LIBSUN= -lsuntool -lsunwindow -lpixrect
LIBIK = -lgik -lik -lgdevserv
MAKE  = /usr/local/contrib/bin/gmake -i -s

Myprog: Myprog.o
    echo "Linking"
    CC Myprog.o -o Myprog $(COOLLIB) -lm

Myprog.o: Myprog.c
    echo "Compiling"
    CC $(cflags) Myprog.c
```

Now the command

make Myprog

will try to compile and link the program.

If you write many programs using this kind of makefile, you will soon find your makefile growing unwieldy. It is possible to define a generic Makefile that will allow you to compile any program by specifying a symbol to make in the command line. For example, in one of my directories I have the following makefile:

```
.SILENT:
CC=CC
COOLBIN= /usr/local/lib
COOLLIB= $(COOLBIN)/cool.a
COOLINC= /usr/local/include/cool
CFLAGS= -c +e1 -I$(COOLINC)
LIBSUN= -lsuntool -lsunwindow -lpixrect
LIBIK = -lgik -lik -lgdevserv
MAKE  = /usr/local/contrib/bin/gmake -i -s

Normal:
    $(MAKE) \
        PROG=$(PROG) CFLAGS="$(CFLAGS)" LIBS="$(COOLLIB) -lm" \
        EXECUTABLE

EXECUTABLE: $(PROG).o
    echo "Linking $(PROG)"
    CC $(PROG).o -o $(PROG) $(LIBS)
```

```
$(PROG).o: $(PROG).c
    echo "Compiling $(PROG)"
    CC $(CFLAGS) $(PROG).c
```

This file can be used to compile and link `Myprog.c` using the command

```
make PROG=Myprog.c Normal
```

The operation of this makefile is fairly clever, so we will describe its main features below.

Line 1 invokes the `.SILENT` attribute which inhibits the printing of each command as it executes.

Lines 2-9 define symbols for use later in the Makefile. Such symbol definitions decrease the size of the Makefile and bring changeable Makefile code to the top where it can be easily found.

The `Normal:` target defines how compiles should "normally" proceed. This command recursively invokes `gmake` with more arguments defining more precisely the kind of compilation to perform. The `Normal:` target basically contains a `make EXECUTABLE` command which is defined below it.

The `make EXECUTABLE` target declares a dependency on `$(PROG).o`; the method for creating that file is described next. The `CC` command uses the `LIBS` symbol to define the libraries to search during linking. This is one of the ways in which other preparation strategies may differ. Programs that require additional libraries, such as `Xlib`, may be compiled under some different target that invokes `make` with a different `LIBS` parameter.

The `$(PROG).o` target performs the actual compilation. Additional modes of compilation may be defined to use different `CFLAGS` parameters to adjust the compilation procedure. For example, debugger flags might be added, or optimizations may be requested by changing the values of the `CFLAGS`.

7.3 Displaytool

The program reproduced next, `displaytool`, supports automatic contrast stretching, display and interaction operations on 2D and 3D diskfiles. `Displaytool` has proven to be one of the most generally useful programs I have written using `COOL`, and the support `COOL` provides is extensive. Unfortunately, I have few intermediate-size programs to demonstrate at present. `Displaytool` grew from a minimal display program to a fairly large one by accretion of needed features. The basic idea is simple. We create a window on the Sun screen with space for 4 images. As we display an image, we apply a linear contrast enhancement so that the full range of gray levels is used. In addition, `displaytool` will scan a file or display an image with 2X sampling (by pixel replication). In polling mode, clicking a mouse button will produce a message telling the viewport, location, and gray level at that location.

The more complex feature supported here is a mask image. If a mask image is used, its gray level is used to determine the color scale in which the intensity of the other image will be displayed. For details, see the User's Manual for the documentation of imagetool.

```
#define D_IMAGETOOL
#define D_IMAGEFILE
#include "coolprelude.h"

imagefile* openimagefile(imagefile* imfilep)
{
    char cmd[80];
    subscript s;
    int plane;
    if(imfilep!=NULL)
    {
        imfilep->filename(cmd);
        cout << "Closing " << cmd << ".\n";
        delete imfilep;
    }
    cout << "Filename: "; cin >> cmd;
    imfilep = new imagefile(cmd);
    imfilep->filename(cmd);
    imfilep->size(s,plane);
    cout << "Image file name: " << cmd;
    char* typ = (char*) imfilep->type();
    cout << "; Type= " << typ;
    cout << "; Size= "; s.print(); cout << " " << plane << "\n";
    return imfilep;
}

image readimage(imagefile* imfilep, int iplane, int autoscale)
{
    double minval=0.0; double maxval=0.0;
    image im;
    if(imfilep==NULL)
    { debug("There is no open imagefile."); }
    else
    {
        cout << "Loading plane " << iplane;
        im = imfilep->load(iplane);
        if(autoscale!=0)
        {
            im.stats0(minval,maxval);
            cout << "; Range = (" << minval << ", " << maxval <<
            ".\n";
            if(minval<0.0 || minval>50.0 || maxval>255.0 ||
            maxval<200.0)
                im.scale(-minval,255.0/(maxval-minval),0.0);
        }
        else
        { cout << ".\n"; }
    }
}
```

```

    }
    return im;
}

void displayimage(int maskflag, imagetool* imtool,
                 image im, image IM, byte enable, const int vp)
{
    if(im.size()==1)
    {
        debug("There is no image to display.");
    }
    else if(maskflag == 1 && IM.size()==1)
    {
        debug("There is no mask to display.");
    }
    else
    {
        int i; i=vp;
        int sel = maskflag;
        while(i<0) {cout << "Viewport:"; cin >> i;}

        i = i%4;
        switch(sel)
        {
            case 0:
                imtool->display(i, im);
                break;
            case 1:
                cout << "Enable byte = " << enable << "\n";
                imtool->display(i, im, IM, enable);
                break;
        }
    }
}

void pollsunwindow(imagetool* imtool, image im, image IM)
{
    int button, viewport=0, imval, IMval, i;
    subscript location;
    debug("Polling sun imagetool window until right button
pressed");
    button=0;
    while(button!=2)
    {
        button=-1;
        imtool->get_input(button, viewport, location);
        if(button >=0)
        {
            imval=im.get_i(location);
            IMval=IM.get_i(location);
            cout << "Button=" << button << " Viewport=" << viewport
<<
                " Location="; location.print();
            cout << " image=" << imval << " mask=";

```

```

        for(i=0;i<8;i++){cout << (IMval&1); IMval= IMval>>1;}
        cout << "\n";
    }
}

```

```

main()
{
    char c,cmd[50];

    debug("Allocate tools");
    imagefile* imfilep = NULL;
    imagefile* IMfilep = NULL;
    image im;
    image IM;
    image* temp1;
    image* temp2;

    debug("Defining flags")
    int contflag=1;
    int maskflag=0;
    int maskops=0;
    int colordisp=0;
    byte enable=255;
    subscript s; subscript t;
    int nplane,i,j;

    debug("Allocating imagetool");
    imagetool* imtool = new imagetool;

    debug("Enter processing loop: ? for help");
    while(contflag)
    {
        cout << "Command(oOrRdDEpsxq?):"; cin >> cmd;
        c = cmd[0];

        switch(c) {

            case 'o': // Open operation
                imfilep = openimagefile(imfilep);
                im = readimage(imfilep,0,1);
                imfilep->size(s,nplane);
                break;

            case 'O': // Open mask operation
                IMfilep = openimagefile(IMfilep);
                IM = readimage(IMfilep,0,0);
                break;

            case 'r': // Read operation
                imfilep->size(s,nplane);
                i=-1;
                while(i<0 || i>nplane-1)

```

```

        {cout << "Plane (0-" << nplane-1 << "):"; cin >> i;}
        im = readimage(imfilep,i,1);
        break;

case 'R':                // Read mask operation
    IMfilep->size(s,nplane);
    i=-1;
    while(i<0 || i>nplane-1)
        {cout << "Plane (0-" << nplane-1 << "):"; cin >> i;}
    IM = readimage(IMfilep,i,0);
    break;

case 'd':                // Display without mask operation
    displayimage(0,imtool,im,IM,enable,-1);
    colordisp=0;
    break;

case 'D':                // Display with mask operation
    displayimage(1,imtool,im,IM,enable,-1);
    colordisp=1;
    break;

case 'E':                // Set plane enable byte for mask
    cout << "Current enable byte = " << enable << "\n";
    cout << "New Enable byte value="; cin >> i;
    enable = i;
    cout << "New Enable byte value=" << enable << "\n";
    break;

case 'p':                // Poll sun window command
    pollsunwindow(imtool,im,IM);
    break;

case 's':                // Scan images

    i = -1;
    while(i<0 || i>nplane-1)
        {cout << "Starting image plane = "; cin >> i;}

    j = -1;
    while(j<0 || j>nplane-1)
        {cout << "Stop at image plane = "; cin >> j;}

    for(;i<=j;i++)
    {
        im = readimage(imfilep,i,1);
        displayimage(0,imtool,im,IM,enable,i);
    }
    colordisp=0;
    break;

case 'x':                // eXpand operation
    temp1 = new image(512,512,BYTE);
    temp2 = new image(512,512,BYTE);

```

```

t=subscript(temp1->shape());
s=subscript(im.shape());

for(t.init(); t.test(); t++)
{
    i=t.r()/2; j=t.c()/2;
    s.set(i,j);
    temp1->set(t,im.get_r(s));
    if(colordisp>0)temp2->set(t,IM.get_r(s));
}
displayimage(colordisp,imtool,*temp1,*temp2,enable,0);
delete temp1;
delete temp2;
break;

case 'q':                // Quit operation
    contflag=0;
    break;

default:
    debug("o: open and read an image file");
    debug("O: open and read a mask image file");
    debug("r: read an image from the current imagefile");
    debug("R: read a mask image from the current mask
imagefile");
    debug("d: display current image in a viewport");
    debug("D: display current image with current mask");
    debug("E: set Enable byte for masks");
    debug("p: poll sun imagetool (to use mouse)");
    debug("s: scan image planes");
    debug("q: quit program");
    break;
}
}
}

```